

# Data Compression Project Final Report

by  
Charles D. Creusere  
Computational Sciences Division  
*Research and Technology Department*  
and  
Jim Witham  
EO/IR Guidance Division  
*Weapons/Targets Department*

OCTOBER 1999

NAVAL AIR WARFARE CENTER WEAPONS DIVISION  
CHINA LAKE, CA 93555-6100



Approved for public release; distribution is  
unlimited.

DTIC QUALITY INSPECTED 4

19991115 048

# Naval Air Warfare Center Weapons Division

---

## FOREWORD

The research described in this report was performed at the Naval Air Warfare Center Weapons Division, China Lake, California, from fiscal year 1996 through fiscal year 1999 as a task under the ONR-funded Air Weaponry Technology Program. This 6.2 program is funded by Code 35 at ONR and is managed locally by Mr. Tom Loftus.

This report was reviewed for technical accuracy by G. A. Hewer.

Approved by  
DR. J. FISCHER, Head  
*Research Department*  
13 October 1999

Under authority of  
C. H. JOHNSTON  
Capt., U.S. Navy  
*Commander*

Released for publication by  
K. HIGGINS  
*Director for Research and Engineering*

## NAWCWD Technical Publication 8442

Published by..... Technical Information Division  
Collation..... Cover, 205 leaves  
First printing..... 61 copies

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, D.C. 20503.				
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE  October 1999	3. REPORT TYPE AND DATES COVERED  Final; fiscal years 1996-1999		
4. TITLE AND SUBTITLE Data Compression Project Final Report (U)			5. FUNDING NUMBERS	
6. AUTHOR(S) Charles D. Creusere Jim Witham				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Air Warfare Center Weapons Division China Lake, CA 93555-6100			8. PERFORMING ORGANIZATION REPORT NUMBER  NAWCWD TP 8442	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT  Public release, distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  (U) The technical objective of this project has been to develop algorithms for compressing digital data that are compatible with the new generation of smart missiles and that can expand the normal operational role of such weapons to include a means for battle damage indication (BDI). A real time BDI system would be an integral and effective tool for adaptive mission planning and retargeting during a strike and for vital imagery required for subsequent rapid retargeting and mission planning. This project has developed the means for compressing video digital data by a large factor—from an almost lossless compression ratio of 20:1 to a highly lossy ratio of 200:1. The compression task also includes compatible coding techniques that convert the compressed video images into an <i>embedded</i> bit stream for datalink transmission. An embedded bit stream is an ordered set of bits representing the compressed image as illustrated by Figure 1-1 (i.e., the bit stream is organized so that the most important bits can be transmitted first). Such embedded coders have many advantages for BDI applications, including their ability to generate fixed-rate bit streams without buffering and their suitability for error-prone communications channels. In order to ensure that the reconstructed BDI images meet the accuracy required by a human or computer interpreter, the encoder has been optimized to achieve the lowest bit rates for a wide range of operating conditions. The compression algorithm has also been designed to operate within the context of a proposed operational system, which must be a low-cost, missile-borne electro-optical (EO) camera or an infrared (IR) imaging array transmitting to a ground station via a datalink. The algorithm and its realization in signal processing hardware has been matched to expected datalink bandwidths and transmission speeds in the year 2000.				
14. SUBJECT TERMS Adaptive mission planning Battle damage indication Compression algorithms			15. NUMBER OF PAGES  408	
Embedded bit stream Fixed-rate bit stream Retargeting			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT  UNCLASSIFIED	18. SECURITY CLASSIFICATION OF ABSTRACT  UNCLASSIFIED	19. SECURITY CLASSIFICATION OF THIS PAGE  UNCLASSIFIED	20. LIMITATION OF ABSTRACT  SAR	

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



## CONTENTS

Acronyms .....	5
Chapter 1. Introduction and Motivation.....	9
Chapter 2. Imaging Platform .....	13
Chapter 3. Image and Video Compression for Remote Sensing .....	21
Summary .....	21
Introduction .....	21
Notation and Terminology .....	22
Image Compression .....	23
Overview.....	23
Discrete Cosine Transform .....	24
Wavelet and Wavelet-Packet Transforms.....	25
Vector Quantization Methods .....	28
Fractal Methods.....	29
Video Compression .....	30
Overview.....	30
Video Encoding and Decoding .....	32
3-D Subband Coding .....	33
Motion-Compensated Interpolation .....	33
Motion Estimation.....	34
WC/BDI Tradeoffs in Video Coders Design .....	34
Encoding Complexity .....	35
Low Latency .....	35
Acceptable Quality .....	36
Flexible Data Types.....	36
Robustness to Channel Errors .....	37
Conclusion.....	38
References .....	38
Chapter 4. Parallel Implementation of Embedded Compression Algorithms.....	43
Summary .....	43
Introduction .....	43
Zerotree Compression Algorithm.....	45

Parallel Wavelet Transforms .....	46
Overview .....	46
Performance Analysis: Overlap-Save Implementations .....	47
Application to Image Compression .....	52
Parallel Implementation of the EZW Coder .....	54
Modified Parallel Implementation .....	60
Performance Comparisons on the TI 320C80 .....	63
Parallel Compression Using Spatial Orientation Trees .....	63
Conclusions .....	64
References .....	65
 Chapter 5. Robustness to Transmission Errors .....	 67
Summary .....	67
Introduction .....	67
EZW Image Compression .....	68
REZW Algorithm .....	69
Basic Approach .....	69
Zerotree Preserving (ZP) Partitioning .....	71
Offset Zerotree (OZ) Partitioning .....	72
Stochastic Analysis .....	73
Results .....	74
Implementation and Complexity .....	79
Conclusion .....	79
References .....	80
 Chapter 6. Interframe Compression .....	 81
Summary .....	81
Introduction .....	81
Conventional Motion Compensation .....	83
Spatial Domain Compensation .....	83
Transform Domain Motion Compensation .....	83
Out-of-Loop Motion Compensation .....	85
Overview .....	85
Practical Implementation .....	87
Video Coding Algorithm .....	94
Motion Estimation .....	94
Transform-Based Residual Encoder .....	95
Results .....	96
Maintaining Robustness to Transmission Errors .....	103
Conclusion .....	104
References .....	105

Chapter 7. Real-Time Intraframe Encoding .....	107
Processor .....	107
Hardware .....	107
Task Partitioning MP Versus PP .....	107
Development Environment .....	107
High Level Description of Routines Used for Intraframe Wavelet Coding .....	108
Chapter 8. Real-Time Intraframe Decoding .....	113
High Level Description of Routines Used for Intraframe Wavelet Decoding .....	113
Chapter 9. Real-Time Interframe Encoding .....	119
Chapter 10. Users Manual for Real-Time Software .....	121
Encode .....	121
Decode .....	122
Chapter 11. Conclusions and Future Work .....	123
Appendixes:	
A. Aerial Platform Analysis .....	A-1
B. Source Code Listings for Real-Time Compression/ Decompression System .....	B-1
C. List of Publications and Patents .....	C-1

(This page intentionally left blank.)

## ACRONYMS

ATM	asynchronous transfer mode
BCH	Bose-Chaudhuri-Hocquenghem
BDI	battle damage indication
bpp	bits per pixel
CD-ROM	compact disk-read only memory
CFD	computational fluid dynamics
CG	center of gravity
CPO	communications partial overlapping
DCT	discrete cosine transform
DMA	direct memory access
DPCM	differential pulse-code modulation
DRAM	dynamic random access memory
DSP	digital signal processor
DWT	discrete wavelet transform
EO	electro-optical
EZW	Embedded Zerotree Wavelet
FFT	Fourier transform
fps	frames per second
HDTV	high definition TV
I/O	input/output
IIR	imaging infrared
IR	infrared
JPEG	joint photography experts group
JSOW	Joint Stand-off Weapon
JTAG	board testing interface
JTIDS	Joint Tactical Information Distribution System
Ladar	laser radar
LBG	Linda, Buzo, and Gray

Mbits/s	megabits per second
MC	motion compensator operator
MIDS	Multifunctional Information Distribution System
MIMD	multiple instruction, multiple data
MLT	modified lapped transform
MPEG	motion picture experts group
MSE	mean squared error
NS	maximum number of scales
NTSC	National TV Systems Committee
OOL	out of loop
ONR	Office of Naval Research
OZ	offset zerotree
PANAIR	panel aerodynamics
PCI	standard PC bus
PE	processing element
P-GPC	periodic generalized pan compensation
PPC	periodic pan compensation
PSNR	peak signal to noise ratio
QAM	quadrature amplitude modulation
RAM	random access memory
REZW	Robust Embedded Zerotree Wavelet
RF	radio frequencies
SAR	synthetic aperture radar
SATCOM	satellite communications
SDRAM	synchronous DRAM
SIMD	single instruction, multiple data
SPIHT	set partitioning in hierarchical trees
TCP/IP	Transmission Control Protocol/Internet Protocol
TI	Texas Instruments
VCR	video cassette recorder
VLSI	very large-scaled integrated
VRAM	video RAM
VQ	vector quantization
VSF	vestigial sideband modulation

WC	weapons control
WT	wavelet transform
ZP	zerotree preserving
ZTR	zerotree-root symbol
3-D	three-dimensional

(This page intentionally left blank.)



## CHAPTER 1.

### INTRODUCTION AND MOTIVATION

The technical objective of this project has been to develop algorithms for compressing digital data that are compatible with the new generation of smart missiles and that can expand the normal operational role of such weapons to include a means for battle damage indication (BDI). A real time BDI system would be an integral and effective tool for adaptive mission planning and retargeting during a strike and for vital imagery required for subsequent rapid retargeting and mission planning. This project has developed the means for compressing video digital data by a large factor—from an almost lossless compression ratio of 20:1 to a highly lossy ratio of 200:1. The compression task also includes compatible coding techniques that convert the compressed video images into an *embedded* bit stream for datalink transmission. An embedded bit stream is an ordered set of bits representing the compressed image as illustrated by Figure 1-1 (i.e., the bit stream is organized so that the most important bits can be transmitted first). Such embedded coders have many advantages for BDI applications, including their ability to generate fixed-rate bit streams without buffering and their suitability for error-prone communications channels. In order to ensure that the reconstructed BDI images meet the accuracy required by a human or computer interpreter, the encoder has been optimized to achieve the lowest bit rates for a wide range of operating conditions. The compression algorithm has also been designed to operate within the context of a proposed operational system, which must be a low-cost, missile-borne electro-optical (EO) camera or an infrared (IR) imaging array transmitting to a ground station via a datalink. The algorithm and its realization in signal processing hardware has been matched to expected datalink bandwidths and transmission speeds in the year 2000.

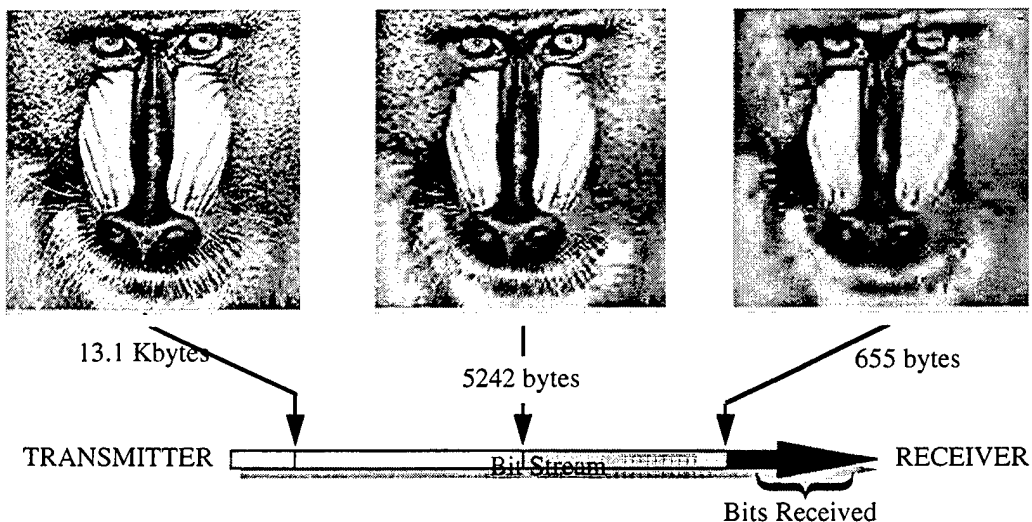


FIGURE 1-1. Embedded Image Compression.

The BDI notional concept is a low-cost, EO camera, or an IR imager, transmitting to a primary receiving station via a datalink. The system definition has three primary components:

1. Imaging sensor requirements
2. Sensor platform requirements
3. Datalink between the sensor platform and a receiving station

Other factors such as weapon speed, warhead size, target set, final attack posture, and expected damage mechanism, along with sensor and platform requirements, have been reviewed and analyzed and a configuration compatible with a Joint Stand-off Weapon (JSOW) submunition pack has been selected. Consider existing and planned military and commercial datalinks:

1. Either a simplified Link 16 protocol (e.g., the surgical strike datalink) channel relayed by aircraft or a commercial Low Earth Orbit data communication service (e.g., Iridium) would be suitable for full motion.
2. Satellite communications (SATCOM) would be acceptable only for still frame transmission.
3. A true Link 16 implementation is too complex.

Because there is no universal metric for the utility of a given data compression algorithm when embedded in a communication system, compression algorithms will be designed within the context of the recommended BDI system. To achieve the very high compression ratios required and yet maximize video throughput and quality, we have implemented the China Lake developed Robust Embedded Zerotree Wavelet (REZW) image compression algorithm on a Texas Instruments (TI) 320C80 multivideo processor. The complete algorithm can precisely achieve any fixed bit rate (i.e., compression ratio) while still delivering state-of-the-art rate-distortion performance (i.e., low distortion at a specified bit rate) or, conversely, achieving any fixed distortion level (up to lossless) over an entire image or just over specific regions. With a single TI 320C80 processor operating at a clock rate of 40 megahertz and a compression ratio of 80:1, our coder maintains a frame rate (with 512x240 frames) of approximately 7 frames per second (fps). At compression ratios of greater than 80:1, the throughput is higher (approximately 15 fps at 200:1) and below 80:1 it is lower (approximately 3 fps at 20:1). However, because the algorithm is highly scalable, its speed increases linearly with the amount of processing power used. For example, we have also implemented the algorithm on a system having two TI 320C80 processors operating at 60 megahertz and have achieved a throughput of 20 fps at a compression ratio of 80:1.

One of our major milestones was the fiscal year 1998 demonstration of our intraframe encoder and decoder operating over an actual communications channel. While a military packet radio channel (e.g., Joint Tactical Information Distribution System (JTIDS) or Multifunctional Information Distribution System (MIDS)) would have been

ideal for this purpose, we did not have access to such equipment at the time nor did we have the financial resources to acquire it. A video broadcast over the internet provides a test bed (1) to study the impact of data packetization on REZW bit stream robustness, (2) to illustrate systems tradeoffs including variable bandwidth, (3) to coordinate battle damage assessment requirements with the BDI rapid retargeting recommendations, and (4) to provide asynchronous transfer mode (ATM) technology directly relevant to Navy needs.

The major challenge in fiscal year 1999 has been to preserve the error resilience of the REZW algorithm in the presence of the temporal differencing required to implement our low complexity interframe coding techniques in real time. We have solved this problem by using a "leaky" prediction model, thereby allowing the decoder to gradually forget errors. The resulting real-time video encoder provides at least twice as much resolution for a given compression ratio while only slightly slowing down the encoder (a 1.5 fps drop at an 80:1 ratio). And even at very high error rates, we have found that the combination of REZW spatial compression and leaky temporal prediction gives excellent error resilience.

In the following chapters, we focus primarily on the technological and scientific contributions that have resulted from this project, along with the real-time demonstration software implementing these concepts. Prior to doing this, however, we set the stage for these discussions by describing a JSOW-compatible BDI imaging platform that was developed and simulated early in this project. Following this, Chapter 3 introduces a variety of image compression algorithms, characterizing their strengths and weaknesses for remote sensing applications like BDI and weapons control. Chapters 4 through 6 discuss our REZW compression algorithm, parallelization of embedded coders for increased speed, and reduced-complexity motion compensated video compression. Chapters 7 and 8 discuss our real-time TI 320C80 implementation and describe the use of this software in detail. Chapter 9 presents our conclusions and future research directions.

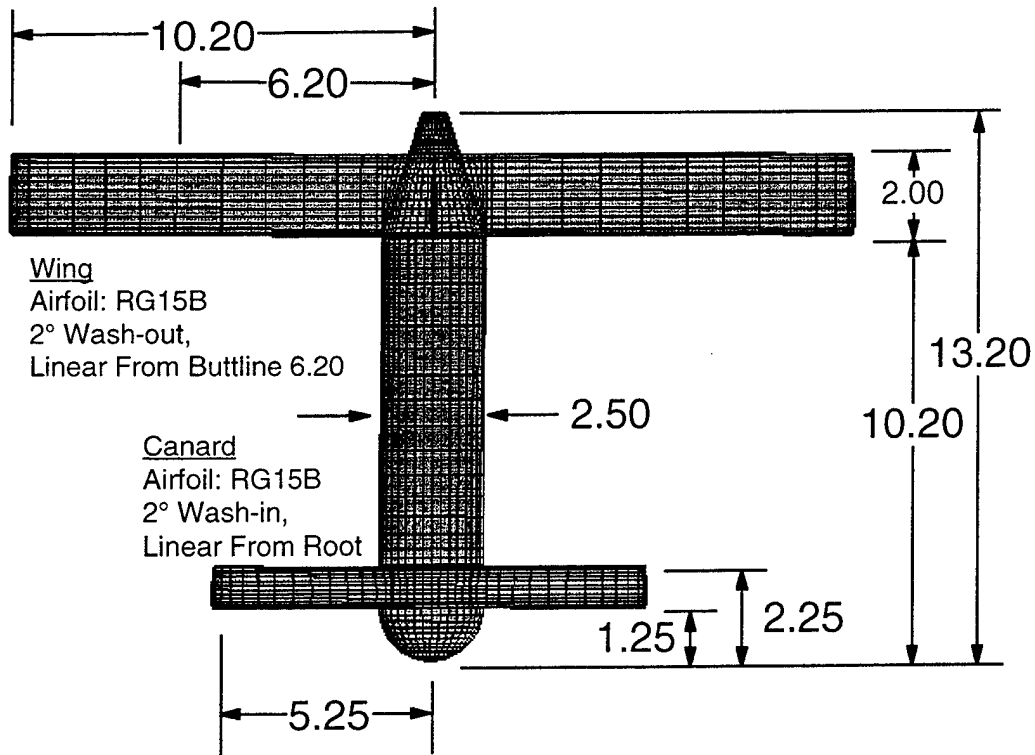
(This page intentionally left blank.)

## CHAPTER 2.

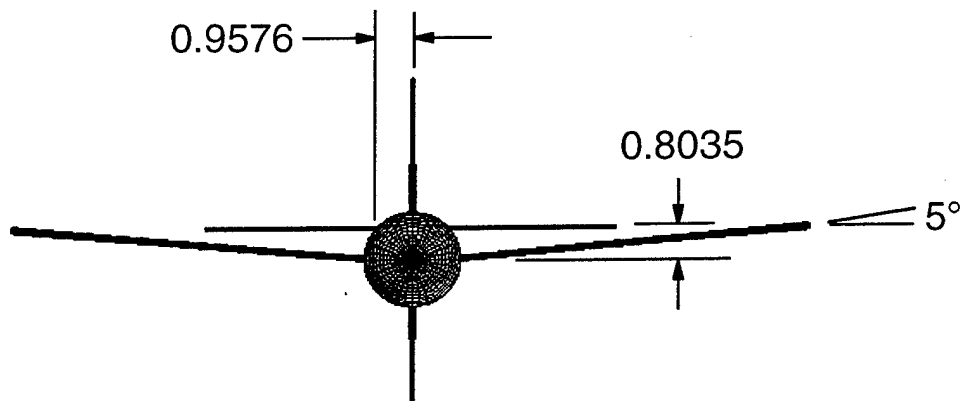
### IMAGING PLATFORM

Early in this project, we investigated a number of ways in which an imaging BDI platform could be incorporated into an existing weapons system. For a variety of reasons, we quickly discarded tethered vehicles and balutes (a cross between a balloon and a parachute). A tethered vehicle did not allow sufficient time to transmit the acquired imagery, while the balute could not be deployed in such a way that it could view the weapon's impact. Instead, we focused on a small glider having collapsible wings, which would replace two submunitions in a JSOW. Unfortunately, aerodynamic analysis showed that this glider was only marginally stable, but through further analysis we determined that flight stability could be increased by adding a small electric motor to the trail of the platform. This JSOW-compatible BDI imaging platform is shown in Figures 2-1 and 2-2.

The vehicle was designed and analyzed aerodynamically using a higher order panel method, PANAIR. This method does not include viscous or separation effects. VSAERO, a low-order panel method, includes an integral boundary layer analysis that provides viscous drag. This value was added to the inviscid drag value obtained from PANAIR. However, this does not give an adequate representation of the viscous flow field about the configuration. Because of its small size and low speed, the vehicle will experience very low Reynolds numbers. Because low Reynolds number aerodynamic properties are very sensitive to geometry and viscous conditions, CFD studies were conducted to aid in their verification. These studies solved the viscous Navier-Stokes equations to model boundary layer growth and separation over the surface of the vehicle, particularly on the boattail and lifting surfaces. These studies considered the effects of the propeller by modeling it as an actuator disk; the actual propeller blades and rotation were not modeled. However, modeling the propeller as an actuator disk was sufficient to understand the effects of the propeller on the vehicle. Appendix A contains the data generated by these simulations.

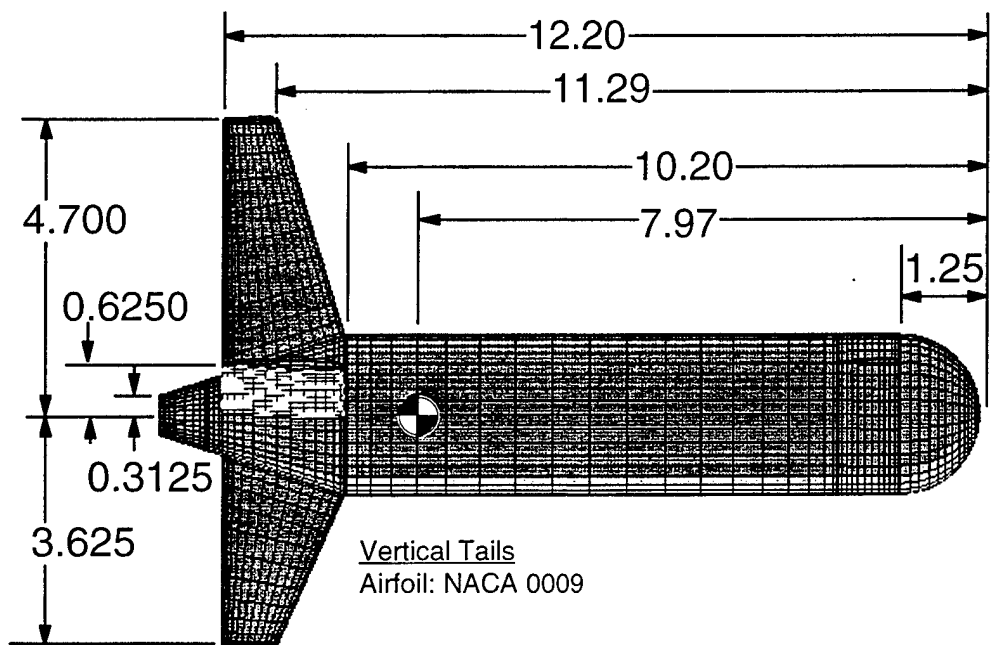


(a) Top view.

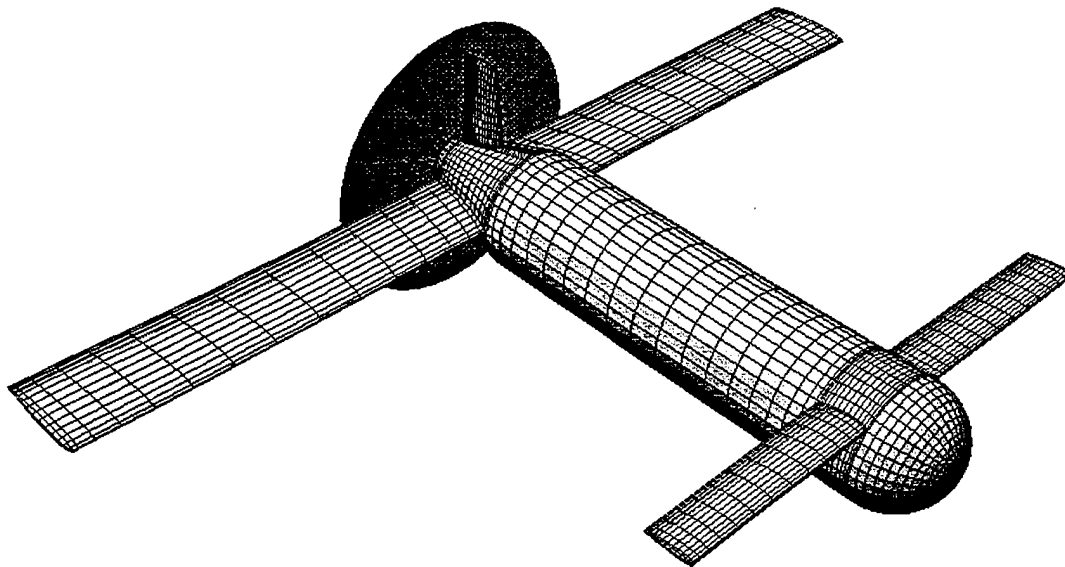


(b) Front view.

FIGURE 2-1. Proposed BDI Platform. Measurements are in inches.



(a) Side view.



(b) Isometric view with actuator disk.

FIGURE 2-2. Additional Views of the Proposed BDI Platform.  
Measurements are in inches.

The lateral aerodynamics of this vehicle could be analyzed using the less detailed PANAIR because solutions are only required with very small sideslip angles. Tailed vehicles, which are laterally stable at small angles of sideslip, will tend to stay laterally stable at larger angles. Thus the sideslip angles necessary to analyze the lateral stability characteristics are on the order of 1 to 3 degrees. If it were considered necessary to analyze the vehicle at larger angles of sideslip or large angles of rudder deflection, it would be necessary to use a computational fluid dynamics (CFD) analysis. However, this is not the case. Figures 2-1 and 2-2 illustrate the tail configuration that was determined to give the best flight characteristics. Note that the vehicle has a vertical surface both above and below the body. Because of the wing/canard configuration, the vehicle's center of gravity (CG) is considerably farther aft than that of a typical flight vehicle. Thus the moment arm is considerably shorter for the vertical tail, which necessitates a greater vertical tail surface area. In addition, because this vehicle is rudder controlled (top vertical tail only), rudder deflections must induce a roll into the turn. This necessitated that the top vertical tail be larger than the bottom and that the wings contain dihedral. This combination affects the flight such that if the rudder deflects the nose to the left, the wings will also bank to the left, initiating a smoothly controlled left turn. Figures 2-3 and 2-4 illustrate the yawing and rolling moments for this configuration. The data show a positive slope for the yawing moment and a negative slope for the rolling moment, which indicate a laterally stable configuration.

The CFD analysis provided both an improved estimate of the basic aerodynamic forces and moments and also a detailed understanding of the flow field about the full configuration. To significantly lower the cost of the analysis, the vertical fins and the left half of the vehicle were not modeled. This significantly lowered the grid generation time and the computer run times, which drastically reduced the cost of the analysis. As a result of the simplified modeling, only longitudinal aerodynamics could be analyzed. But this was sufficient because lateral aerodynamics can be predicted adequately using PANAIR, which is considerably less costly. Force and moment results of the PANAIR and CFD analyses are presented in Figures 2-5 through 2-8.

Based on this and other analysis, the proposed image platform appears to be both stable and controllable while delivering sufficient loitering time above the target area. However to be completely sure, one must also perform wind tunnel tests to validate the simulation results. Because of budget constraints, we were not able to perform such tests.



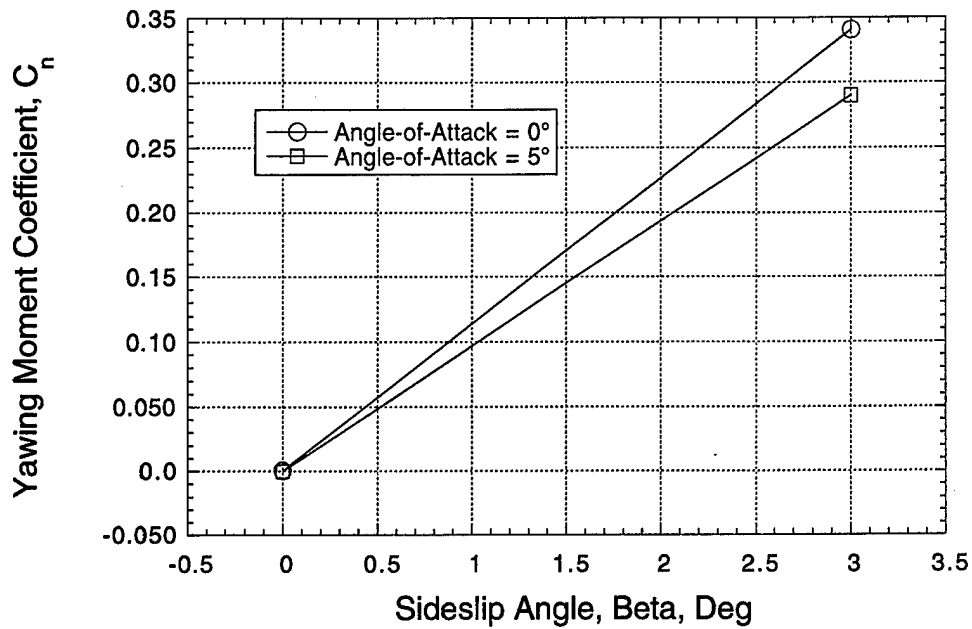


FIGURE 2-3. PANAIR Solutions of Yawing Moment Coefficient for AOAs of 0 and 5 Degrees.

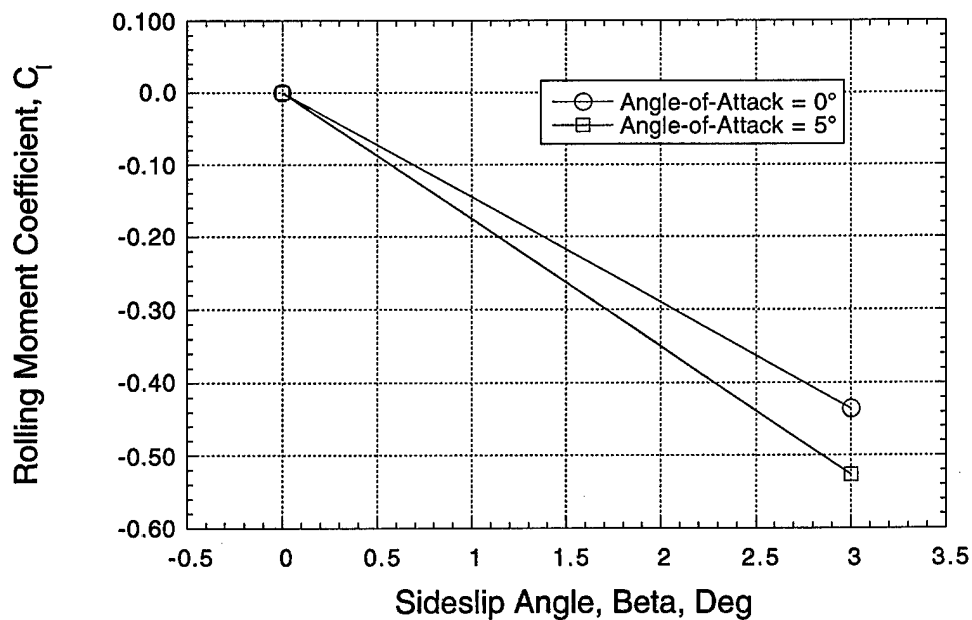


FIGURE 2-4. PANAIR Solutions of Rolling Moment Coefficient for AOAs of 0 and 5 Degrees.

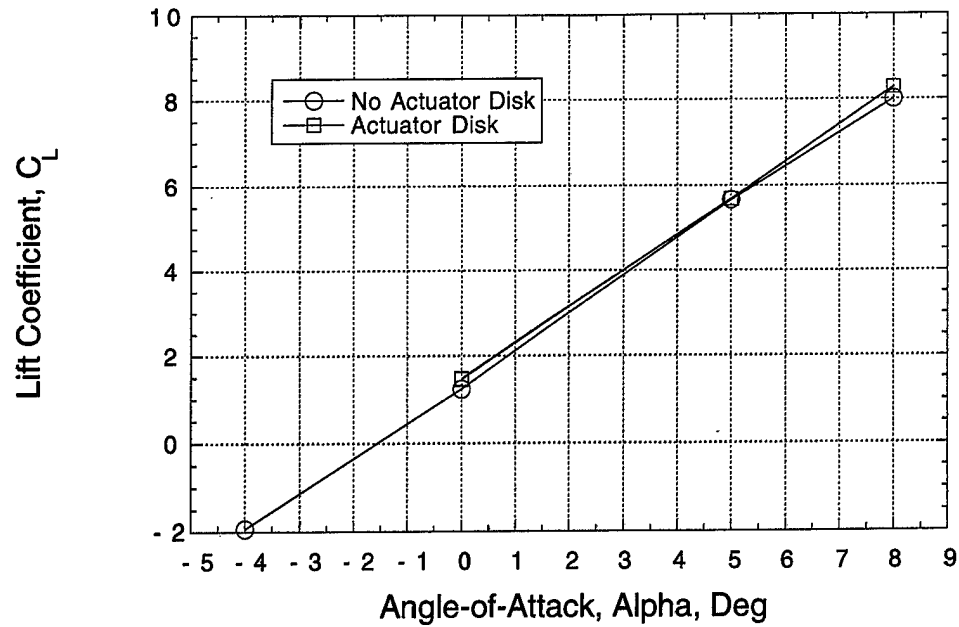


FIGURE 2-5. Lift Coefficient Comparison of the Navier-Stokes Solutions With and Without the Actuator Disk at Sea Level, With Turbulent Boundary Layer.

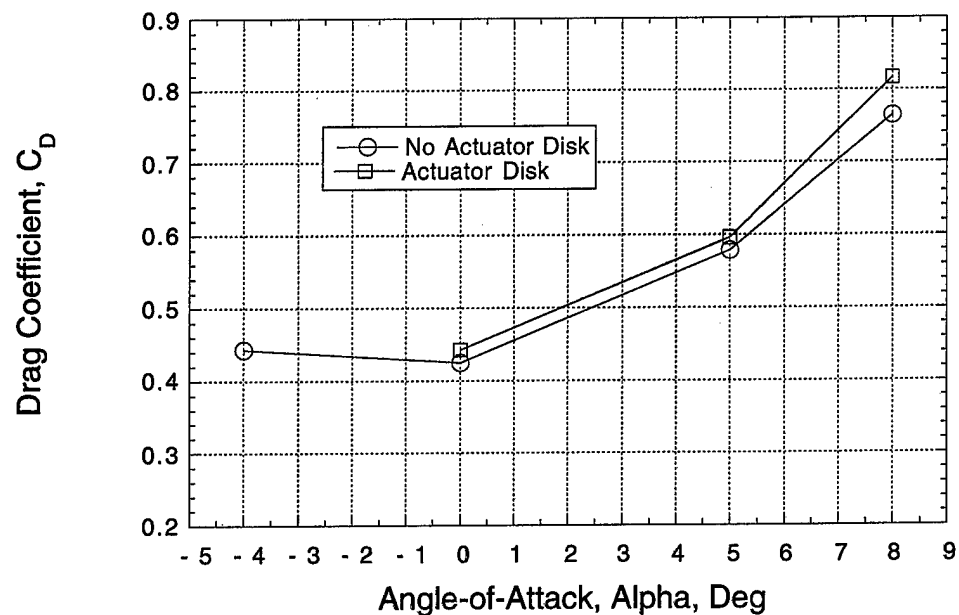


FIGURE 2-6. Drag Coefficient Comparison of the Navier-Stokes Solutions With and Without the Actuator Disk at Sea Level, With Turbulent Boundary Layer.

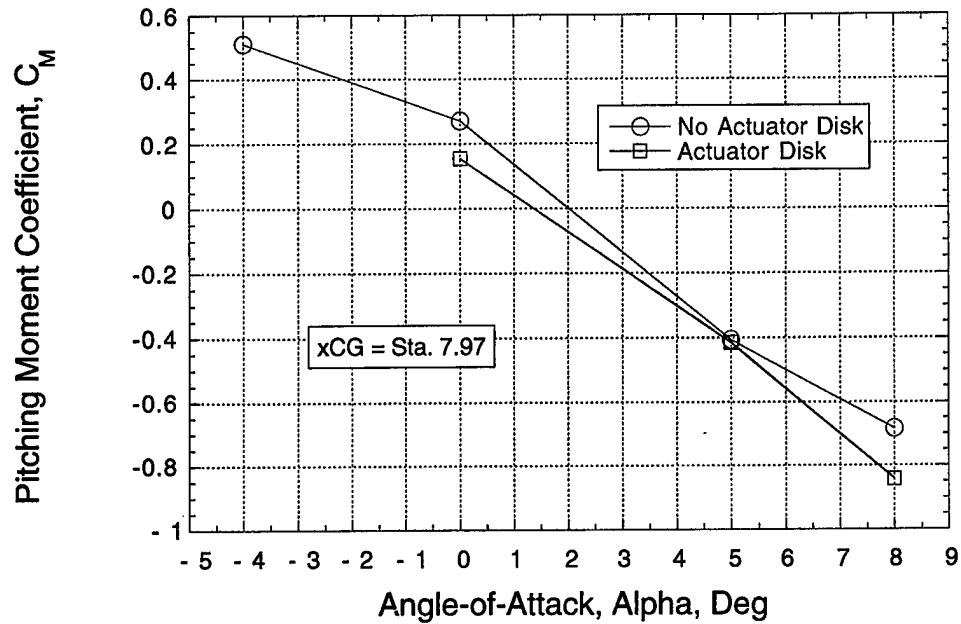


FIGURE 2-7. Pitching Moment Coefficient Comparison of the Navier-Stokes Solutions With and Without the Actuator Disk at Sea Level, With Turbulent Boundary Layer.

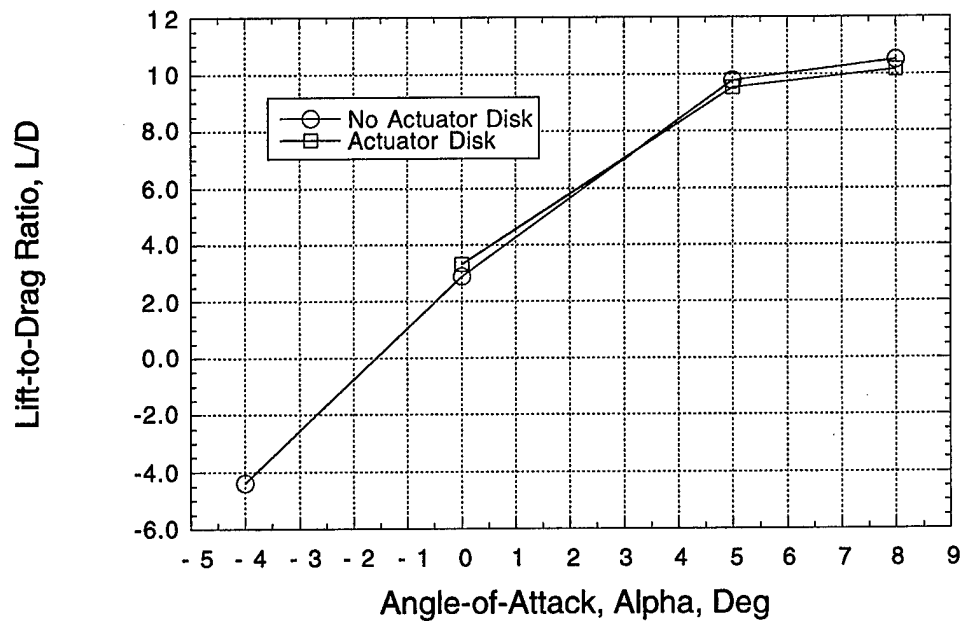


FIGURE 2-8. Lift-to-Drag Coefficient Comparison of the Navier-Stokes Solutions With and Without the Actuator Disk at Sea Level, With Turbulent Boundary Layer.

(This page intentionally left blank.)

## CHAPTER 3.

### IMAGE AND VIDEO COMPRESSION FOR REMOTE SENSING

#### SUMMARY

The goal of this chapter is to familiarize the reader with the major image and video compression technologies in the context of weapons control (WC) and BDI applications. Toward this goal, we discuss the operation and limitations of the four major still image compression techniques: discrete cosine transform (DCT), wavelet transform (WT), vector quantization, and fractal. We then consider the broader problem of implementing video compression algorithms, which use the previously discussed still image compression algorithms as basic building blocks. Some of the techniques examined here are motion estimation, motion compensation for prediction and interpolation, and three-dimensional (3-D) subband coding. Finally, we discuss the major characteristics of digital video compression and transmission systems, concentrating on the impact they have on WC and BDI system performance. Many of these characteristics are desirable in every compression application (e.g., good image quality) but some of them are especially important for low-cost, remote-sensing platforms (e.g., a low-complexity video encoder). In addition, some characteristics such as video latency are important for the WC application but not for the BDI application. Within this framework, we discuss the various image and video compression technologies available and highlight those that are particularly good or bad for WC/BDI applications.

#### INTRODUCTION

Recent advances in computational hardware and signal processing theory have made the transmission of real-time digital video possible. Thus far, there have been two driving applications in this field: (1) video broadcast via such media as CD-ROM (compact disk – read only memory), terrestrial radio frequencies (RF), and satellite RF; and (2) video teleconferencing. The key paradigm for video broadcasting, which is epitomized by the MPEG standards (References 3-1 and 3-2), is to make the receiver as cheap as possible because there are many receivers but only a few transmitters. For video teleconferencing, on the other hand, the goal is to make the system as symmetric as possible because every system must be able to both transmit and receive video in real time. In addition, broadcast video is generally expected to be of very high quality (motion picture experts group (MPEG)) 1 video should look at least as good as that of a standard VCR (video cassette recorder), for example), while a considerable amount of degradation is accepted in teleconferencing video.

The paradigms for military remote-sensing video are considerably different from those of either broadcast or teleconferencing video. First and foremost, the transmitter and, consequently, the video encoder, must be as low in complexity as possible, while the decoder complexity is far less important. Reducing the complexity of the encoding translates into reduced space, weight, and power requirements on the sensor platform. At the very least, relaxing these requirements reduces the cost of the complete system, but it can also be the enabling factor that makes the system possible in the first place. The issue of transmitter complexity and its associated costs becomes even more important for WC/BDI applications, where the transmitter is expendable. Currently, analog video datalinks derived from the National TV System Committee (NTSC) television broadcast standard are used, but these suffer from a variety of shortcomings including low resolution, jamming/intercept vulnerability, and a lack of flexibility to varying video data types (e.g., laser radar (Ladar), synthetic aperture radar (SAR), imaging infrared (IIR), etc.). Digital video transmission can overcome all of these shortcomings, albeit at a higher transmitter cost.

In this paper, we study a number of compression technologies and evaluate their suitability for WC/BDI applications. Below we introduce the notation and terminology used throughout this paper, and then discuss the four most common still frame compression technologies. Still frame compression is studied both for its own sake and because it forms the basis of the video compression strategies also discussed in this chapter. We then analyze the ability of the various algorithms presented in the earlier sections to satisfy the WC/BDI compression requirements. Finally, conclusions are presented.

## NOTATION AND TERMINOLOGY

Throughout this technical paper, the terms *compression* and *coding* are synonymous and will be used interchangeably. We use the word *coding* to describe the compression process because an actual bit stream describing the image is output by the compression algorithm. Historically, this process is referred to as *source coding*, a term which originates from Shannon's original work on information theory (Reference 3-3). Consistent with this terminology, we refer to the part of the algorithm that performs compression as the *encoder* and the part that performs decompression as the *decoder*.

In general, there are two forms of coding, lossless and lossy. As the names suggest, *lossless* coding implies that the image produced by the decoder is identical to that which went into the encoder. Unfortunately, the compression ratio that one can achieve using lossless coding is image dependent and is generally not more than 4:1 (4 times fewer bits in the compressed image than were in the original). *Lossy* coding, on the other hand, implies a loss of information in the coding process, leading to a degradation in the quality of the decoded image. To accurately characterize the quality of a lossy coder, one must

specify a rate-distortion curve (i.e., the bit rate (compression ratio) of the coded image, versus the distortion introduced by the coding process). This distortion is typically measured by either mean squared error (MSE) defined as

$$\text{MSE} \equiv \frac{1}{X \cdot Y} \sum_{m=0}^{X-1} \sum_{n=0}^{Y-1} |x(m,n) - \hat{x}(m,n)|^2 \quad (3-1)$$

or peak signal to noise ratio (PSNR), which is generally given in decibels and is defined by

$$\text{PSNR} \equiv 20 \cdot \log_{10} \left( \frac{255^2}{\text{MSE}} \right) \quad (3-2)$$

In Equation 3-1, the original and coded images are given by  $x(m,n)$  and  $\hat{x}(m,n)$ , respectively, and both are  $X$  by  $Y$  pixels in size. Thus, in later sections when we discuss the *rate-distortion performance* of a coding algorithm, we are simply referring to the average distortion introduced into the output image, as measured by Equations 3-1 and 3-2, compared to the number of bits used to represent it in coded form.

## IMAGE COMPRESSION

### OVERVIEW

The objective of still image compression or coding is to reduce the number of bits of information needed to represent a given image by eliminating redundancy in the image (lossless compression) and by introducing distortion into the image in a manner that is acceptable to the viewer (Reference 3-4). While lossless compression is desirable, the amount of bit rate reduction it can achieve depends on the entropy or information content of the image. Allowing distortion to be added to the image by the compression process results in a loss of information, but it also makes it possible to achieve arbitrarily high compression ratios at the cost of accepting large amounts of distortion. Thus any lossy image compression algorithm must be characterized by a rate versus distortion curve in order to properly quantify its quality.

In recent years, four major coding techniques have dominated the engineering literature, and all of these have become commercially available in some form. The four major approaches are DCT, WT, vector quantization (VQ), and fractal. Some of these techniques can also be combined to create a hybrid coding algorithm; the most common approach combines a transform with VQ. In this work, we give only a brief overview of each of these methods, focusing on characteristics that affect their utility in WC/BDI applications.

## DISCRETE COSINE TRANSFORM

Coders based on the DCT are by far the most pervasive today. Included in this group are the MPEG 1 and 2 standards for broadcast video (References 3-1 and 3-2), the CCITT H.261 standard for video teleconferencing (Reference 3-5), and the joint photographic experts group (JPEG) standard for still image compression (Reference 3-6). For these coding algorithms, a separable 2-D DCT is performed on each 8x8 block of image pixels. Specifically, the forward transform of the DCT is defined by the JPEG standard as

$$F(u, v) = \frac{C(u)C(v)}{4} \sum_{j=0}^7 \sum_{k=0}^7 \left\{ f(j, k) \cos \left[ \frac{(2j+1)u\pi}{16} \right] \cdot \cos \left[ \frac{(2k+1)v\pi}{16} \right] \right\} \quad (3-3)$$

where  $f(j,k)$  is an 8x8 block of image pixels and

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases} \quad (3-4)$$

The inverse transform is similarly defined with the roles of  $(j,k)$  interchanged with those of  $(u,v)$ . Using these relatively small blocks reduces the amount of computation required and preserves a reasonable amount of spatial locality in the transform domain. This second point is important because the underlying statistics of a typical image are highly nonstationary and therefore, to maximize coding efficiency, bits must be allocated spatially across the image in a nonuniform manner. In addition, because non-overlapping blocks of the image are transformed, the entire process is easily parallelized. However the use of non-overlapping blocks is also the Achilles heel of such algorithms at low bit rates because block boundaries begin to appear in the reconstructed image. These artifacts begin to emerge in JPEG-coded images as the compression ratio is increased above 20:1.

Figure 3-1 is a block diagram of the complete transform-based image encoder in its simplest form. In the figure, each block of the image is first converted into its frequency components by the transform (defined by Equation 3-3) for the DCT). After transformation, the blocks of coefficients are quantized, with varying numbers of bits allocated to different frequency components. The distribution of bits among the frequency components can be either fixed for all time or can be adapted and transmitted as side information; this second approach generally provides superior performance. After quantization, the coefficients are losslessly encoded either by using run-length and Huffman coding or by using an arithmetic coder. This lossless coding will eliminate any remaining statistical redundancy in the image; thus if the corruption of 1 bit during transmission causes the reconstructed image to degrade into white noise, the encoder has done an excellent job. It should be noted that the use of lossless coding makes this algorithm inherently variable in bit rate. Thus getting the compressed image through a fixed bit rate channel requires either that we iterate the coding process (changing a



quality factor until the desired bit rate is achieved) or that we use a large buffer with some form of feedback control from the quantizer. This latter approach is the one taken in the MPEG standards.

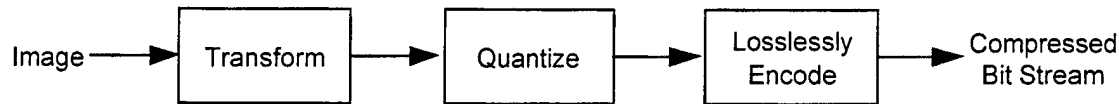


FIGURE 3-1. Image Encoder Block Diagram.

In the WC/BDI problem, DCT-based methods have much to offer. First and foremost, the complexity of a still-frame DCT-based encoder is very low because the complexity of the 8x8 DCT is very low. In addition, the algorithm scales linearly with the dimensions of the image, and it is highly parallelizable. Finally, a great deal of available commercial hardware can be easily leveraged for military applications. The only real drawback of a DCT-based coding algorithm is its performance at low bit rates: the boundaries between the 8x8 blocks appear in the reconstructed image. Consequently, for low bit rate transform coding, transforms that maintain spatial locality and yet have overlapped blocks are preferable. Such transforms are often called modified lapped transforms (MLTs) or multirate filter banks (Reference 3-7), and certain members of this general class can be used to efficiently implement discrete wavelet and wavelet-packet decompositions.

## WAVELET AND WAVELET-PACKET TRANSFORMS

Wavelet and wavelet-packet transforms are two members of the general class of subband coding methods. While other forms of subband coding have been discussed in the literature at great length (Reference 3-8), a general consensus has formed over the past few years that wavelet and wavelet-packet techniques offer the best in coding performance.

Figure 3-2 is the block diagram of one level of a 2-D wavelet transform, where L and H indicate lowpass and highpass filters, respectively, and the subscripts v and h indicate the direction in which the filter operates (vertical or horizontal). The downward-pointing arrow followed by the 2 represents the downsampling operation in which every other sample point is discarded (this maintains the same sampling density in the wavelet domain as existed in the original image). To get a true wavelet transformation, the decomposition of Figure 3-2 is iteratively applied, first to the original image and then to the low-low band at each successive level until, in the limiting case, there is only one pixel remaining in the final low-low band. Figure 3-3 shows a mapping of wavelet coefficients for a 3-level wavelet decomposition, where the two capital letters indicate the frequency band (e.g., HL corresponds to a high-low band) and the subscript indicates the scale of that band. In this figure, lower scale numbers correspond to finer edge features in

the image, while higher numbers correspond to subsampled coarse features. Note that all of the bands represent different resolutions of edge features except the final low-low band, which contains all of the remaining low frequency image content. The wavelet coefficient mapping illustrated in Figure 3-3 can be converted perfectly back into the original image if the correct filters  $L$  and  $H$  are used in Figure 3-2. The conditions on these filters were first presented in Reference 3-9 and later used by Daubechies in her compactly supported (i.e., having a finite impulse response filter bank implementation) orthogonal wavelets (Reference 3-10). In order to show the equivalence between this discrete wavelet transform (DWT) and the continuous wavelet transform, Daubechies required her filters to have an additional quality called regularity, which is somewhat related to the number of zeros that filter  $L$  has in the frequency domain at  $\pi$ . Relaxing the orthogonality constraint, perfect reconstruction biorthogonal wavelets and filter banks have also been created (Reference 3-11), and these have been shown in recent years to be the most effective for image compression applications (Reference 3-12).

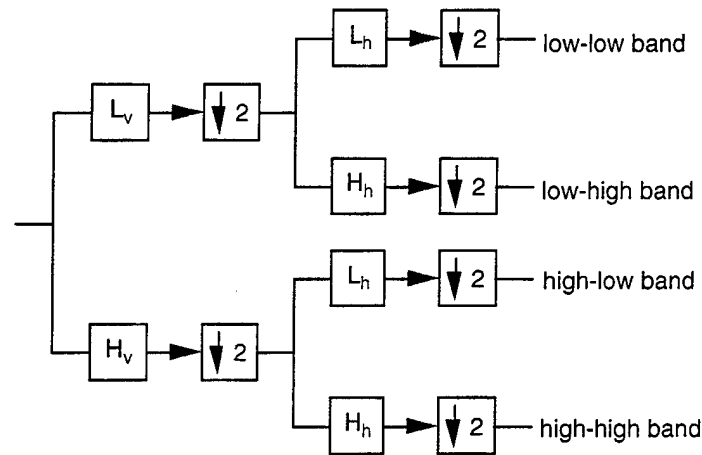


FIGURE 3-2. Four-Band, 2-D Wavelet Decomposition.

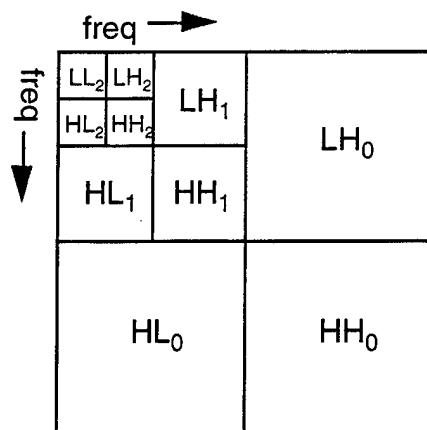


FIGURE 3-3. Wavelet Coefficient Map.

Early wavelet compression algorithms followed the same format as the DCT-based coder illustrated in Figure 3-1: transform, quantize, and losslessly encode (Reference 3-13). This methodology generated better results with a wavelet transform than with a DCT, but the difference was only on the order of a few decibels in PSNR improvement. It was not until the advent of the Embedded Zerotree Wavelet (EZW) algorithm developed by Shapiro (Reference 3-14) that a vast improvement in still frame compression quality was realized over DCT-based methods like JPEG. The most important concept introduced in this algorithm is that of forming zerotrees of wavelet coefficients. This idea makes it possible to efficiently exploit the correlation between insignificant coefficients at different scales. This basic concept has since been further optimized using an entropy-constrained quantization approach to achieve some of the best wavelet coding results published to date (Reference 3-15).

A wavelet-packet decomposition uses the basic filtering process described by Figure 3-2, but instead of successively subdividing the low-low band, it adapts the decomposition to the image using some criterion. The first work in this field used entropy to determine whether a given band was further subdivided (Reference 3-16), while later work optimized the decomposition in a rate-distortion sense (Reference 3-17). These methods are very computationally expensive compared to a standard wavelet transform because of the added need to adaptive the decomposition, but when combined with the zerotree concept, they deliver excellent rate-distortion performance on difficult images (Reference 3-18).

In terms of the objective PSNR (or equivalently MSE) distortion measure, wavelet-based coding algorithms outperform DCT-based algorithms at all bit rates. This difference, however, only becomes visually perceptible at lower bit rates and even this is mostly because of the serious blocking artifacts that appear in the DCT-coded image. Thus for higher bit rate applications (e.g., less than 20:1 compression ratios), there is little perceptible advantage to using a wavelet-based coder versus a DCT-based coder, but there is a considerable increase in computational complexity. The computational complexity of a DWT for an  $N \times N$  image is  $kN^2$ , where the constant  $k$  depends on the exact filters used in Figure 3-2 and the number of levels of the wavelet decomposition. For the very low complexity 5/3 biorthogonal wavelet (Reference 3-11) with 5 levels of decomposition,  $k$  is 30; while for the allpass wavelet (the most computationally efficient wavelet having good filter quality) it is 7 (Reference 3-19). The complexity of the block-based DCT, on the other hand, is also  $kN^2$  but with  $k$  less than 1, giving it a considerable computational advantage. At very low bit rates, however, the perceptual quality of a wavelet-coded image is far better than that of a block-based DCT. In such situations, the image coded with the DWT is generally blurred, suffering from ringing or blocking artifacts at sharp edges within the image. The exact nature and extent of these artifacts depend on the lengths and smoothness of the wavelet filters used in the decomposition. For shorter biorthogonal wavelets, such distortions are not severe, and reasonable image quality can often be attained down to compression ratios of 100:1.

## VECTOR QUANTIZATION METHODS

A Vector Quantizer groups  $L$  input samples (in this case pixels) together into an input vector  $\mathbf{x}$  and measures the Euclidean distance,

$$d = \sum_{k=0}^{L-1} (x(k) - c_i(k))^2 \quad (3-3)$$

between that vector and a set of codevectors,  $c_i$ , in what is called the codebook. The index  $i$  of the codevector with the smallest Euclidean distance (i.e., the one most similar to the input vector) is then transmitted to the receiver, which then looks up this index in its own codebook to find an approximation to the original input vector. The encoder and decoder for this process are illustrated in Figure 3-4. Note that the encoder and decoder must have access to exactly the same codebooks and that the quality of this method is directly dependent on the quality of the codebook.

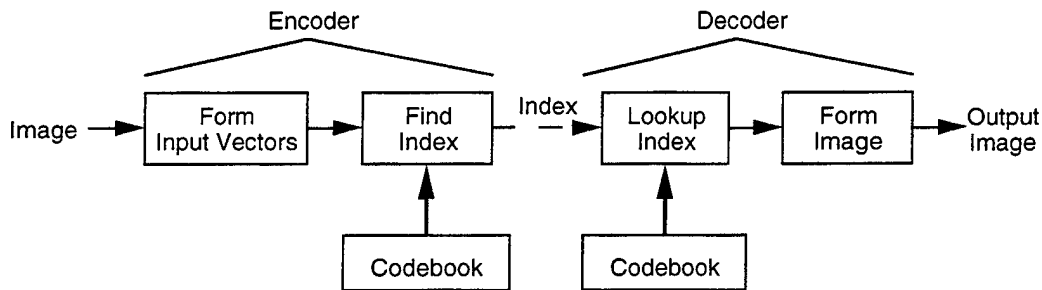


FIGURE 3-4. Vector Quantization-Based Encoder and Decoder.

The motivation for this technique comes from an observation in Shannon's landmark paper (Reference 3-3) that by allowing the size of the coded vector to grow to infinity, the efficiency of the coder could achieve the theoretical bound (i.e., the vector could be coded with the absolute minimum number of bits). Unfortunately, infinite length vectors are difficult to work with in practice, and this theoretical observation provides no clues as to how one might actually design the vector quantizer. An algorithm called the generalized Lloyd or LBG (Linda, Buzo, and Gray) has been developed to design VQ codebooks, but it is only guaranteed to converge to a locally optimal solution (Reference 3-20). As the dimension of the input vector and the size of the codebook increase, it becomes more and more difficult to find good solutions using this algorithm. This problem becomes especially obvious when one directly applies VQ to image compression because the input vectors are reasonably large (typically 4x4 or 8x8 groups of pixels) and the codebooks are very large (many thousands of codevectors). Even worse, however, is the huge variety of possible input sequences. The VQ codebook design process tries to distill out of a sequence of training vectors the essential "truths" of that sequence (i.e., the fundamental feature blocks that best describe a set of images). Unfortunately, the space of all possible image sequences is simply too broad for such a method to be effective. On smaller, narrowly defined image sets, though, such techniques may prove to be very useful.

In order to overcome some of the limitations of this unstructured VQ approach, many researchers have studied ways to add structure to the VQ coder. Some approaches that have been found to be useful for image compression include mean-gain-shape VQ (Reference 3-21), transform VQ (References 3-22 and 3-23), product code VQ (Reference 3-24), and multiresolutional VQ (Reference 3-25). All these techniques allow *a priori* knowledge about the input source to be included into the coding model, simplifying and robustifying the design. In addition, if the VQ is forced to have a binary, tree-structured codebook, the search complexity for a codebook of size  $M$  can be reduced from  $M$  Euclidean distance comparisons to just  $\log_2 M$ . The decoder operation is simply a table lookup operation, so the encoder search represents all of the complexity in a conventional VQ implementation. From the perspective of the WC/BDI problem, it is unfortunate that most of the complexity resides in the expendable encoder. If vectors of size  $4 \times 4$  are used, each Euclidean distance calculation requires 32 operations, making the total complexity  $2 \cdot M \cdot N^2$  operations for an image of size  $N \times N$ . While this is quite high ( $M$  is generally greater than 1,000), the structure of this search process is very regular and, consequently, well suited to very large-scaled integrated (VLSI) implementations.

Despite their theoretical advantages, the performance of most VQ coder algorithms on images taken from outside their training sets lags that of transform-based coders. It should be noted, however, that vector quantization is actually a superset of all of the other image coding techniques discussed in this paper. For example, a coder with identical rate-distortion performance to the JPEG coder can be implemented using a full search VQ on  $8 \times 8$  blocks of image pixels, with the DCT, quantization, and lossless compression implicitly incorporated into the VQ codebook. Even fractal coding, which is discussed below, can be implemented in a VQ framework. Of course, it is generally more efficient to implement transform and fractal coders explicitly than it is to use a VQ formulation.

## FRACTAL METHODS

Fractal coding methods are based on the assumption that natural images have fractal characteristics (i.e., that the same basic features appear in different spatial locations and at different scales throughout the image). While a fractal coder also uses transformations, it is far different than those discussed above because the transforms themselves and not the transform coefficients are what is transmitted to the decoder. A fractal-based encoding algorithm is shown in Figure 3-5. Basically, a number of mappings must be devised that map all of the blocks of the image to other, smaller blocks of the image. Going from large to small blocks results in a contractive mapping, which is a necessary condition to ensure convergence in the decoder when the mapping is iterated. Overlapping is allowed in both the domain (original) blocks and the range (translated and contracted) blocks; the Collage Theorem is used to smoothly fuse blocks together. If the mappings are designed properly, an approximation of the original image can be reconstructed in the decoder by iteratively implementing these mappings (starting from any arbitrary image) until convergence is achieved. An excellent, in-depth treatment of fractal coder design is presented in (Reference 3-26).

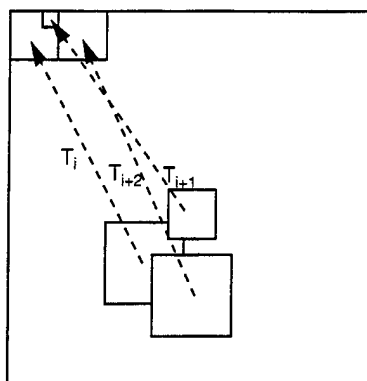


FIGURE 3-5. Block Mappings for an Iterative Fractal Encoder.

While fractal coding algorithms are among the most mathematically elegant, they do have their drawbacks. The first and foremost is that their encoder complexities are extremely high. By contrast, the complexity of a fractal decoder is almost trivial, but, unfortunately, this asymmetry is exactly the opposite of what is needed for still frame compression in WC/BDI applications. In addition, the rate-distortion performance of fractal coders has been found to be generally inferior to that of DCT- and wavelet-based coders. Some impressive claims of good image quality at compression ratios of 1000:1 have been made, but these have not been substantiated in comparative tests on large, diverse sets of natural images.

## VIDEO COMPRESSION

### OVERVIEW

In video coding algorithms, both interframe and intraframe redundancies are removed from an image sequence. To remove intraframe redundancy, one of the still image compression algorithms discussed above is often used. The most common video coding schemes use block-based motion-compensated prediction to create a residual image that has very low energy and can, therefore, be coded with very few bits. A generic encoder is shown in Figure 3-6 and the corresponding decoder is shown in Figure 3-7. Most of the video coding algorithms using motion compensation fit into this framework, including MPEG 1 and 2. Comparing the figures, one notes that the video encoder actually contains a complete copy of the still frame decoder in its feedback loop. Excluding the still frame encoder and decoder, the two most complex blocks in Figure 3-6 are the motion estimation and rate buffer blocks. By effectively estimating the motion and compensating for it in the feedback loop, a residual image with little energy is formed; this can be coded by the still frame encoder with very few bits. As a general rule, however, the better the motion estimate is, the more complex the estimator must be. The

rate buffer, on the other hand, does not require a lot of computational processing power to implement: it is complex because it must dynamically adjust the quantization in the still frame encoder so that a constant bit rate can be output without buffer over- or underflows. Performing this operation well requires large amounts of hand optimization in the design process and a lot of logic circuitry in the buffer controller. A number of possible rate buffer implementations have been proposed (References 3-27, 3-28, and 3-29).

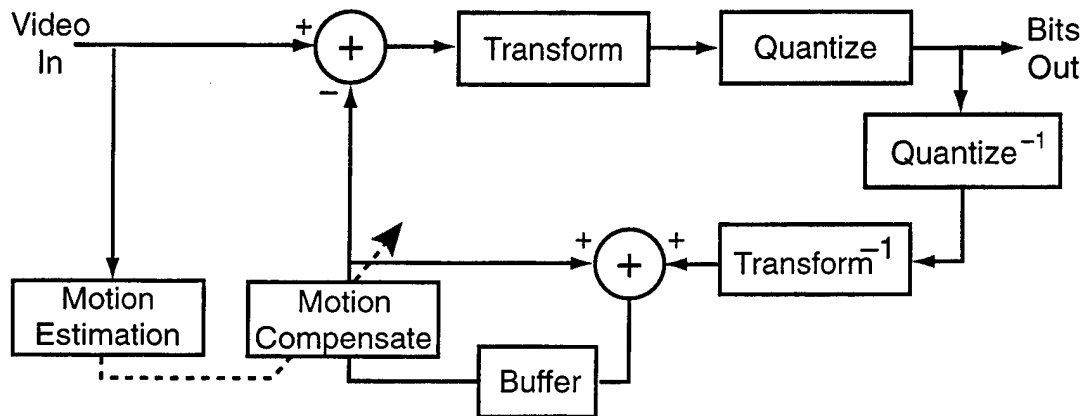


FIGURE 3-6. Generic Hybrid DPCM-Transform Video Encoder.

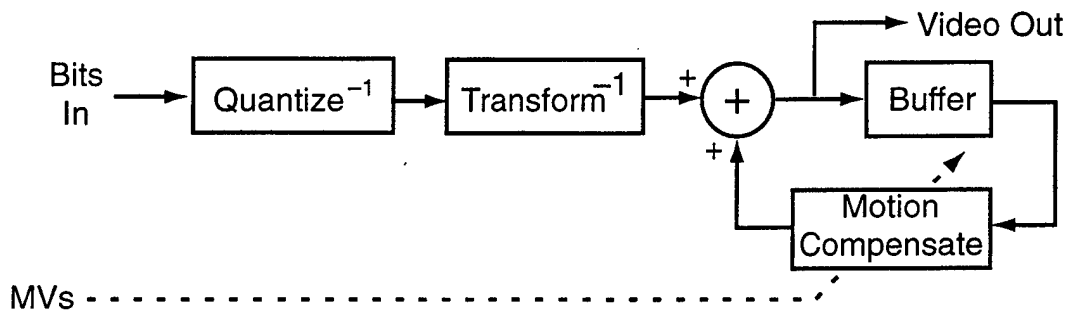


FIGURE 3-7. Generic Hybrid DPCM-Transform Video Decoder.

## VIDEO ENCODING AND DECODING

The video encoder described by Figure 3-6 can produce three different types of video frames: I-frames, B-frames, and P-frames. A typical mix of such frames is shown in Figure 3-8. An I-frame is coded using only information from within that frame (i.e., intraframe coded), and, consequently, allows a clean break-in or resynchronization point within the video sequence. It is very important to have these break-in points to allow the receiver to resynchronize in the event of an uncorrected transmission bit error. The density of I-frames determines how many seconds of video are corrupted in the event of a transmission error. A P-frame, on the other hand, is the prediction residual from the last P- or I-frame in the sequence: it has no dependence on future frames. Obviously, P-frames do not offer break-in points, and errors from past frames will propagate into them. The last standard frame is a B-frame, which is bidirectionally predicted from both past and future frames. The highest compression is achieved with B-frames, but they cannot be used to predict future frames because they, themselves, depend on future frames.

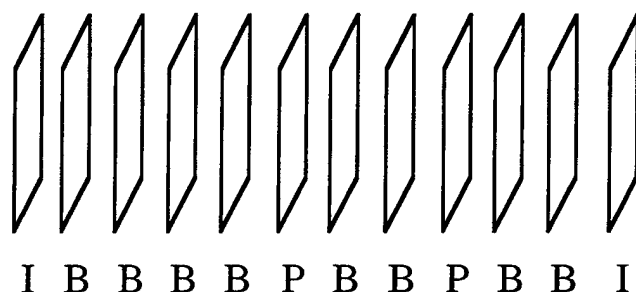


FIGURE 3-8. Typical Mix of Coded Frames. I-frames, P-frames, and B-frames are intra, unidirectionally-predicted, and bidirectionally-predicted frames, respectively.

By comparison, the operation of the video decoder as shown in Figure 3-7 is quite simple, with its most complex element being the still frame decoder. A frame reordering block is needed only if bidirectionally predicted frames are used and, while it increases the memory requirements of the decoder, it has no impact on its computational complexity. Note that the decoder shown in Figure 3-7 does not include motion-compensated interpolation. Using such interpolation greatly increases the computational complexity of the decoder with no cost increase in the encoder, making it well suited to the remote-sensing paradigm.



### 3-D SUBBAND CODING

Another video compression technology showing much promise in recent years is motion-compensated 3-D subband coding. A 3-D subband decomposition transforms the sequence not only in the two spatial dimensions but also in the temporal dimension, creating a time-frequency-space mapping of the sequence. In theory, this decomposition alone should be able to capture and localize motion information, but recent work has found that the addition of global motion compensation greatly improves performance (References 3-30 and 3-31). The video encoding and decoding algorithms shown in Figures 3-6 and 3-7 also describe the basic structure of a motion-compensated 3-D subband coder, where the "image encoder" and "image decoder" blocks now process sets of images. Obviously, this forces other blocks in the system, most notably the motion compensation block, to process sets of image frames as well.

### MOTION-COMPENSATED INTERPOLATION

An important technique for improving the perceptual quality of compressed image sequences is bidirectionally motion-compensated interpolation, illustrated in Figure 3-9. The past and future frames are reconstructed, and then these are used along with motion information to interpolate the missing frames. In Figure 3-9, for example, the interpolated frame  $I_1$  is a linear combination of four-fifths of the past frame and one-fifth of the future frame, where each of these have been motion compensated before being combined. To perform the motion compensation, the past frame is moved forward along the motion vector(s), while the future frame is moved backward along it (them). If multiple vectors are used in a block-based compensation scheme, overlapping and non-covered areas in the interpolated images must be considered. Bidirectional interpolation does not truly add any new information to the image sequence, but it does increase its perceived quality by maintaining a sharp, smooth blending of true image frames, simulating a far higher frame rate than was actually transmitted. Note that bidirectional interpolation is a part of the bidirectional predictive coding used in Figures 3-6 and 3-7: the bidirectionally interpolated image forms the estimate, which is subtracted from the original image to create the differential update (B-frame) in the encoder. The MPEG coding algorithm also allows for the use of motion-compensated interpolation in the decoder (Reference 3-1).

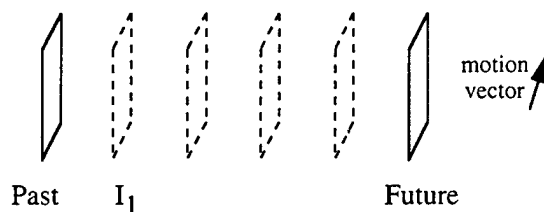


FIGURE 3-9. Motion Compensated Interpolation.

## MOTION ESTIMATION

The motion estimation process is truly the heart of video compression in that it has a very large effect on the efficiency of the system and on the quality of the resulting output. The most common method currently used is block-based motion estimation, where blocks of pixels from the past frame are compared with blocks in the current frame in order to determine where they moved. Using these vectors, one can then compensate the past frame in Figure 3-6 for the motion (e.g., move the blocks to their new locations) before subtracting it from the current frame (Reference 3-32). The most common way to determine the movement of the blocks is to use full-search block matching, in which a spatial-domain correlation is performed between a block in the previous frame and a region around that block in the current frame. This is computationally expensive and it often results in poor motion estimates when displacements of less than one pixel are involved. To overcome this second problem, one can interpolate the blocks to achieve subpixel accuracy, but this further increases the computational complexity. Faster spatial correlation methods such as the three-point search have been developed to reduce the computational complexity (Reference 3-4), but these only converge to the optimal motion vector if the error surface is convex (which is not often the case).

Another class of motion vector estimation methods are those based on frequency domain correlations. For example, block matching can also be done by taking the fast Fourier transforms (FFTs) of the past and current blocks, multiplying them, and then taking the inverse FFT. This has the advantage over spatial domain correlation in that subpixel accuracy can be achieved simply by zero padding in the transform domain. Another approach is to perform the frequency domain correlation over the entire frame and then do block matching at the correlation spikes to determine which blocks moved where. This has proven to be especially effective for high quality interpolation using a variant frequency-domain method called phase correlation (References 3-33, 3-34, and 3-35). Finally, we note that, thus far, pixel based methods like optical flow (References 3-36, 3-37, and 3-38) have not been widely applied in image coding because of the high cost associated with coding the flow vectors.

## WC/BDI TRADEOFFS IN VIDEO CODERS DESIGN

The most important features in a video coding algorithm designed for WC and BDI applications are low encoding complexity, low latency, acceptable quality, flexible data type, and robustness to channel errors. All of these can be simultaneously achieved as long as one does not also constrain the bit rate. Of course, all realistic channels have bandwidth constraints, so perfectly satisfying all of these constraints in the same system is impossible. Instead, one must study the trade-offs associated with different design decisions.

## ENCODING COMPLEXITY

### Still Frame Encoding

The conventional video encoder shown in Figure 3-6 contains a still frame encoder and decoder. In the case of a DCT- or wavelet-based algorithm, the inclusion of this feedback loop doubles the complexity of the video encoder. Using instead a non-symmetric still frame scheme like VQ or fractal coding greatly reduces the cost of adding the decoder to the feedback loop but greatly increases the basic cost of the encoder, leading to a net increase in computational complexity over a transform-based system. Given this, along with our earlier discussion about still-frame coding algorithms, we conclude that the best choice is a transform coder, although which one depends very much on the bit rate at which the system is designed to operate. Because both the forward and inverse transforms must be performed, the computational advantage of the block-based DCT over the wavelet transform becomes even more pronounced. On the other hand, if too few bits are allocated to code each I-frame, severe artifacts are introduced into the sequence by the transform, and these will only be exacerbated in the differentially coded frames. Consequently, despite its computational complexity, wavelet-based coders might be preferable at lower bit rates.

### Video Encoding

As pointed out previously, the feedback loop in Figure 3-6 adds a great deal of computational complexity to the system. If subpixel interpolation is used in the motion-compensation block, the encoder complexity is increased even more. If the motion compensation can instead be performed in the transform domain, the still frame decoder block can be eliminated, greatly reducing the complexity. While this has been recently proposed (Reference 3-39), the idea is not yet proven. Better still would be the elimination of the entire feedback loop, but this would make it much more difficult to exploit temporal redundancy in the sequence.

## LOW LATENCY

The need for low latency is most obvious in the weapons control problem: if the delay between when the image was acquired and when it is viewed by the pilot becomes too great, then by the time the pilot's control input gets to the weapon it might be too late to update the weapon's course. The latency issue is simplified, however, because the pilot need not directly control the weapon: he might simply perform the final target selection. Thus a few seconds of latency in the video may be acceptable, as long as a frame motion history is maintained in the weapon (i.e., the weapon knows where a spot designated in a past frame has moved to in the current frame). The issue of exactly how much latency can be tolerated for weapons control then depends largely on the speed and maneuverability of the weapons as well as the speed of the target. Because man-in-the-

loop control is generally only used for ground targets and with relatively slow weapon platforms, it is likely that considerable delay can be tolerated.

For the BDI problem, the requirements for video latency are less severe, but they still exist. Consider, for example, an encoder that introduces a latency of 2 seconds into the video stream. With such an encoder we will lose, on the average, the last second of video, which if the BDI sensor is mounted on the weapon itself, could be critical information. Thus latency must also be considered in this application.

## ACCEPTABLE QUALITY

While objective measures such as MSE (or, equivalently PSNR) roughly characterize image quality, it is well known that they have many flaws. It is for this reason that human perceptual evaluations have formed the cornerstone of high definition TV (HDTV) evaluation. The same holds true for WC/BDI applications, but with a twist: it is not important that the video "looks good," but rather it is only important that the video be usable. Thus pilots and analysts must be able to examine the video to recognize targets and to assess damage. A highly compressed image appears to be blurrier than the original, but the actual distortion is highly nonlinear. This means that subjective evaluations with the actual users of this video are necessary—evaluations that must be performed within a statistically valid framework to be meaningful. In other words, the evaluation must do more than simply asks for opinions: it must assess the ability of the subject to correctly perform his/her job. Little such data currently exist, making it impossible to accurately characterize the image quality required of a future WC/BDI system.

## FLEXIBLE DATA TYPES

Logistically, it would make sense to use common, scalable image and video compression algorithms for all resolutions and types of sensors. Visible and IIR sensors are the most likely candidates for WC/BDI compression, but one might wish to use the same datalink to also transmit Ladar and SAR images for other applications. Thus, it makes sense to design flexibility into the system. In a general sense, just using a digital communications system greatly increases the flexibility over older analog systems that had frame rates and resolutions built directly into the waveform. To achieve the best compression performance on all data types, however, it is important that the encoder be fully adaptive (i.e., that it not depend on *a priori* knowledge about the image set). Such knowledge is invariably sensor dependent and is unlikely to generalize well. This *a priori* information often appears in a coding algorithm in the form of VQ codebooks, nonuniform scalar quantizers, Huffman lookup tables, and arithmetic coder models. These last two are only a problem if they are fixed rather than image adaptive.

## ROBUSTNESS TO CHANNEL ERRORS

The RF environment is a noisy one because every electronic device radiates some RF energy at some frequencies. In the military arena, this unintentional radiation is compounded by the problem of intentional jamming by the enemy. The process of compression or "source coding," as it is called by Shannon (Reference 3-3), extracts redundancy from the input signal. If the source coding is optimal (i.e., all redundancy has been extracted) then a single bit error in transmission will turn the reconstructed signal at the receiver into noise. To protect the compressed signal, it is necessary to add redundancy back into it in such a way that errors can be detected and corrected. This is called "channel coding." Shannon has shown that the optimal transmission signal (in terms of bit rate and robustness to channel errors) can be achieved by using separate source and channel coding (Reference 3-3). The two most common forms of channel coding used today are the block-based Reed-Solomon and BCH codes and the infinitely extending convolutional codes (Reference 3-40). Recently, research has begun on joint source-channel coding in an effort to decrease the complexity of the complete system, but such efforts are still in their early stages (Reference 3-41).

The other major factor affecting both the bit throughput and the signal robustness is the analog modulation. Using advanced modulation techniques like quadrature amplitude modulation (QAM, Reference 3-42) or vestigial sideband modulation (VSB, Reference 3-43), it is possible to robustly transmit 20 megabits per second (Mbits/s) of digital data over a single 6-megahertz analog television channel. Even using MPEG 1 compression with its nominal bit rate of 1.5 Mbits/s, one can transmit 13 digital channels in the space currently used for one analog channel. These advanced modulation schemes are optimized for Gaussian noise because that is generally a reasonable model for noise sources in such RF channels. Thus, while a Gaussian noise jammer is very hard to overcome for an analog video transmission system, it is actually the best case for a digital system. For the digital system, long burst errors are the worst case, but this problem can be minimized by scrambling the bit stream as it leaves the channel coder. Complete diagrams of the transmitter and the receiver are shown in Figures 3-10 and 3-11.

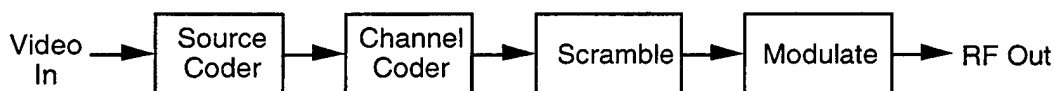


FIGURE 3-10. Complete Video Transmission System.

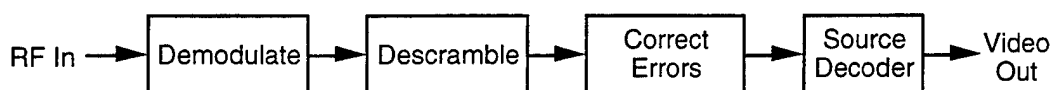


FIGURE 3-11. Complete Video Receiver.

Unfortunately, little work has been done so far in characterizing interference sources in the military environment and quantifying their effects on the quality of digitally transmitted video. It is quite possible that unintentional RF radiation from the aircraft could be a greater problem than intentional jamming from the ground. Ultimately, one or more video transmission systems must be rigorously demonstrated in a realistic RF environment before these questions can be convincingly answered.

## CONCLUSION

We have discussed a wide range of image and video coding algorithms, analyzing their suitability for WC and BDI applications. In the course of this investigation, we have concluded that while block-based DCT algorithms are superior for high bit rate, high quality video applications, wavelet-based algorithms are the better choice for low bit rate, reduced-quality video. Before actual systems can be adopted, however, there are still many questions to be answered. Of these, the two most important are what level of video quality is required for the WC and BDI applications and how does the battlefield RF environment affect the video datalink. Once these questions have been answered, the problem of constructing a new digital video compression and communications system will be bounded, allowing the many engineering trade-offs associated with such a system to be made and ultimately resulting in the introduction of this technology onto the battlefield.

## REFERENCES

- 3-1. D. J. LeGall. "The MPEG Video Compression Algorithm: A Review," *Image Proc. Algorithms and Techniques II*, SPIE Vol. 1452 (1991), pp. 444-457.
- 3-2. K. Challapali et al. "The Grand Alliance System for US HDTV," *Proc. of the IEEE*, Vol. 83, No. 2 (February 1995), pp. 158-174.
- 3-3. C. E. Shannon. "A Mathematical Theory of Communications," *Bell Systems Technical Journal*, Vol. 27 (1948), pp. 379-423, 623-656.
- 3-4. A. N. Netravali and B.G. Haskell. *Digital Pictures: Representation and Compression*. NY, Plenum Press, 1988.
- 3-5. "Video Coder for Audio Visual services at px64 Kbits/sec," Recommendation H.261, The Consultative Committee on International Telephony and Telegraphy, 1990.

- 3-6. G. K. Wallace. "Overview of the JPEG (ISO/CCITT) Still Image Compression Standard," *Image Proc. Algorithms and Techniques*, SPIE Vol. 1244 (1990), pp. 220-233.
- 3-7. P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ, Prentice Hall, 1993.
- 3-8. J. W. Woods and S. D. O'Neil. "Subband Coding of Images," *IEEE Trans. on Acoustics, Speech, and Signal Proc.*, Vol. ASSP-34 (October 1986), pp. 1278-1288.
- 3-9. M. J. T. Smith and T. P. Barnwell. "Exact Reconstruction Techniques for Tree-Structured Subband Coders," *IEEE Trans. on Acoustics, Speech, and Signal Proc.*, Vol. ASSP-34, No. 3 (June 1986), pp. 434-440.
- 3-10. I. Daubechies. "Orthonormal Bases of Compactly Supported Wavelets," *Comm. on Pure and Applied Math.*, Vol. XLI (1988), pp. 909-996.
- 3-11. I. Daubechies. *Ten Lectures on Wavelets*, Philadelphia, PA, SIAM, 1992.
- 3-12. M. Lightstone and E. Majani. "Low Bit-Rate Design Considerations for Wavelet-Based Image Coding," *Proc. of the SPIE Symposium on Visual Comm. and Image Proc.* (September 1994).
- 3-13. M. Antonini and others. "Image Coding Using Wavelet Transforms," *IEEE Trans. on Image Proc.*, Vol. 1, No. 2 (April 1992), pp. 205-220.
- 3-14. J. M. Shapiro. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. on Signal Proc.*, Vol. 41, No. 12 (December 1993), pp. 3445-3462.
- 3-15. Z. Xiong, K. Ramchandran, and M. T. Orchard. "Joint Optimization of Scalar and Tree-Structured Quantization of Wavelet Image Decompositions," *Proc. of the 27th Annual Asilomar Conf. on Signals, Systems, and Computers* (November 1993), pp. 891-895.
- 3-16. R. R. Coifman and M. V. Wickerhauser. "Entropy-Based Algorithms for Best Basis Selection," *IEEE Trans. on Info. Theory*, Vol. 38, No. 2 (March 1992), pp. 713-718.
- 3-17. K. Ramchandran and M. Vetterli. "Best Wavelet Packet Bases in a Rate-Distortion Sense," *IEEE Trans. on Image Proc.*, Vol. 2, No. 2 (April 1993), pp. 160-175.

- 3-18. Z. Xiong, K. Ramchandran, M.T. Orchard, "Wavelet Packet-Based Image Coding Using Joint Space-Frequency Quantization," *Proc. Int. Conf. on Image Proc.* (November 1994), pp. 324-328.
- 3-19. C. D. Creusere and S. K. Mitra. "Image Coding Using Wavelets Based on Perfect Reconstruction IIR Filter Banks," *IEEE Trans. on Circuits & Systems: Video Technology*, Vol. 6, No. 5 (October 1996), pp. 447-458.
- 3-20. Y. Linde, A. Buzo, and R. M. Gray. "An Algorithm for Vector Quantizer Design," *IEEE Trans. on Communications*, Vol. COM-28 (January 1980), pp. 84-95.
- 3-21. M. Lightstone and S. K. Mitra. "Entropy-Constrained Mean-Gain-Shape VQ for Image Compression," *Proc. of the SPIE VCIP* (1994).
- 3-22. P. H. Westerink and others. "Subband Coding of Images using VQ," *IEEE Trans. on Communications*, Vol. 36, No. 6 (June 1988), pp. 713-719.
- 3-23. M. Antonini and others. "Image Coding Using VQ in the Wavelet Transform Domain," *Proc. Int. Conf. on Acoustics, Speech, and Signal Proc.* (1990), pp. 2297-2300.
- 3-24. M. Lightstone, D. Miller, and S. K. Mitra. "Entropy-Constrained Product Code VQ With Application to Image Coding," *Proc. Int. Conf. on Image Proc.* (November 1994).
- 3-25. A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, Boston, MA, 1992.
- 3-26. A. E. Jacquin. "Fractal Image Coding: A Review," *Proc. of the IEEE*, Vol. 81, No. 10 (October 1993), pp. 1451-1465.
- 3-27. C. T. Chen and A. Wong. "A Self-Governing Rate Buffer Control Strategy for Pseudoconstant Bit Rate Video Coding," *IEEE Trans. on Image Proc.*, Vol. 2, No. 1 (January 1993), pp. 50-59.
- 3-28. M. R. Pickering and J. F. Arnold. "A Perceptually Efficient VBR Rate Control Algorithm," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 527-532.
- 3-29. K. Ramchandran, A. Ortega, and M. Vetterli. "Bit Allocation for Dependent Quantization With Application to Multiresolution and MPEG Video Coders," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 533-545.
- 3-30. J.-R. Ohm. "Three-Dimensional Subband Coding With Motion Compensation," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 559-571.



- 3-31. D. Taubman and A. Zakhor. "Multirate 3-D Subband Coding of Video," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 572-588.
- 3-32. M. Bierling. "Displacement Estimation by Hierarchical Blockmatching," *Proc. SPIE Visual Comm. Image Proc.*, Vol. 1001 (1988), pp. 942-951.
- 3-33. M. Goetze. "Generation of Motion Vector Fields for Motion Compensated Interpolation of HDTV Signals," *Signal Proc. of HDTV*. Elsevier Science Publishers, North-Holland, 1988, pp. 383-391.
- 3-34. G. M. X. Fernando and D. W. Parker. "Motion Compensated Display Conversion," *Signal Proc. of HDTV*. Elsevier Science Publishers, North-Holland, 1988, pp. 393-399.
- 3-35. Th. Reuter and H. D. Hoehne. "Motion Vector Estimation for Improved Standards Conversion," *Signal Proc. of HDTV*. Elsevier Science Publishers, North-Holland, 1988, pp. 345-354.
- 3-36. J. L. Barr, D. J. Fleet, and S. S. Beachemin. "Systems and Experiments, Performance of Optical Flow Techniques," *Int. Journal of Computer Vision*, Vol. 12, No. 1 (1994), pp. 43-77.
- 3-37. G. Hewer, C. Kenny, and W. Kuo. "Wavelets, Curvature, and Chaining Issues With Application to Optical Flow," *Proc. of the SPIE Conf.*, Vol. 2277 (July 1994), pp. 253-264.
- 3-38. G. Hewer, C. Kenney, W. Kuo, and L. Peterson. "Curvature and Aggregate Velocity for Optical Flow," *Proc. of the SPIE Conf.*, Vol. 2567 (July 1995).
- 3-39. H. Ito and N. Farvardin. "On Motion Compensation of Wavelet Coefficients," *Proc. Int. Conf. on Acoustics, Speech, and Signal Proc.* (May 1995).
- 3-40. R. E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley Pub. Co., Reading, MA, 1983.
- 3-41. "Gaussian Channels," *Proc. Int. Conf. on Acoustics, Speech, and Signal Proc.* (May 1995).
- 3-42. L. Goldberg. "IC Opens 500-Channel Frontier to Cable Systems," *Electronic Design*, Vol. 42, No. 23 (7 November 1994), pp. 71-80.
- 3-43. R. Citta and others. "The Digital Spectrum Compatible HDTV Transmission System," *IEEE Trans. on Consumer Electronics*, Vol. 38 (August 1991), pp. 101-107.

(This page intentionally left blank.)

## CHAPTER 4.

### PARALLEL IMPLEMENTATION OF EMBEDDED COMPRESSION ALGORITHMS

#### SUMMARY

Here we consider the problem of parallelizing the basic wavelet transform by studying two methods of accomplishing this that are compatible with inter-scale coding techniques. Both of these methods use an overlap-save approach: one performs all data transfers before computation begins, while the other requires that data be exchanged between processors as the wavelet computations are proceeding. We study these two methods to determine the conditions under which each provides the optimal solution. Next, we present a multiple instruction, multiple data (MIMD) parallel partitioning of a complete image compression algorithm based on Shapiro's EZW concept (Reference 4-1). The rate-distortion penalties associated with this straightforward parallelization are thoroughly quantified. Noting that the performance of this parallel algorithm might be unacceptably poor for large processor arrays, we discuss an alternate single instruction, multiple data (SIMD) algorithm, which achieves the same rate-distortion performance as the sequential EZW algorithm at the cost of higher complexity and reduced scalability. Finally, we examine an improved image compression algorithm developed by Said and Pearlman (Reference 4-2) and show how it can also be efficiently implemented in parallel using the approach developed for the EZW coder.

#### INTRODUCTION

In the last few years, wavelet-based image and video compression algorithms have been developed that are greatly superior to those based on the DCT in terms of both objective criteria (bit rate versus distortion) and subjective criteria (Reference 4-1 through 4-4). Still, DCT-based coders have two important advantages over wavelet-based coders: lower computational complexity and a trivial parallel implementation. Both of these advantages are achieved because the 2-D DCT is implemented independently on non-overlapping 8x8 (or 16x16) blocks of image pixels. The use of non-overlapping blocks, however, is also the cause of the DCT-based coder's most notorious defect: at low bit rates, blocking artifacts appear in the reconstructed image.

In contrast, wavelet-based algorithms do not suffer from such artifacts, making them a better choice for low bit rate image and video coding applications. While one might consider using different filters to reduce the complexity of the wavelet transform (see Reference 4-5), one can never match the simplicity of the DCT. We note, however, that

the problem of real-time video coding using wavelets is not truly one of raw computational complexity but rather one of throughput; the wavelet-based coder must be able to keep up with the sequence of video frames. While it might be possible to solve this problem using a single, very fast processor, it is probably more efficient with today's high levels of VLSI integration to use an array of slower processors, each working on a portion of the problem. The efficiency of such a parallel solution hinges on two factors: the complexity of the operations performed by the individual processors and the amount of communications required between them. In this work, we consider the problem of efficiently parallelizing compression algorithms that exploit inter-scale redundancy, focusing primarily on Shapiro's EZW algorithm (Reference 4-1) and extending work previously presented in Reference 4-6. Our baseline parallel system is an array of processing elements (PEs) with 2-D mesh interconnections, as shown in Figure 4-1. Unless otherwise noted, we assume that each of these PEs is an asynchronous MIMD device with its own memory (e.g., a distributed memory architecture). While many other authors have considered the general problem of parallelizing wavelet transforms (References 4-7 through 4-11), we study it here in the context of a specific coding algorithm and illustrate how the complete system can be implemented in an efficient manner.

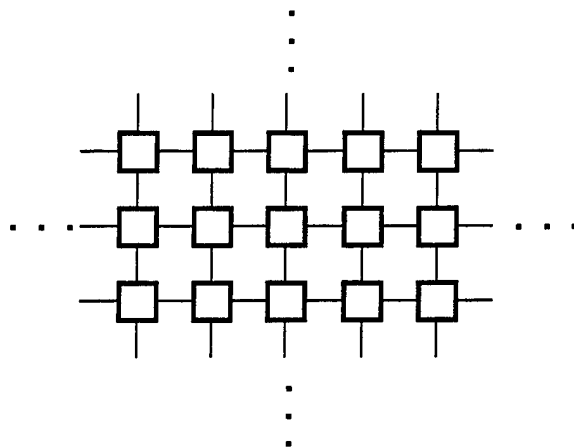


FIGURE 4-1. A Mesh-Interconnected Processor Array.

We briefly discuss the sequential EZW image compression algorithm below, and then we consider parallel implementations of the basic wavelet transforms. The main goal here is to select an efficient parallelization that is compatible with the coding algorithm, and, to this end, we consider two variations of the overlap-save approach. We then present results quantifying the rate-distortion penalties associated with different levels of encoder parallelism. We will also introduce a modified architecture that eliminates these rate-distortion penalties, at the expense of added hardware requirements and reduced scalability. Then we discuss the general applicability of our parallel partitioning approach to the Said and Pearlman image compression algorithm. Finally, conclusions are presented.

## ZEROTREE COMPRESSION ALGORITHM

Because a major goal of this work is to study the trade-offs associated with parallelizing EZW-based image compression, we first present a brief discussion of the basic sequential algorithm. The fundamental observation around which this coding algorithm is centered is that there is a strong correlation between insignificant coefficients at the same spatial locations in different wavelet scales. (i.e., if a wavelet coefficient at a coarser scale is zero, then it is more likely that the corresponding wavelet coefficients at finer scales will also be zero). Figure 4-2 is a 3-level, 2-D wavelet decomposition and the links that define a single zerotree structure. If the wavelet coefficient at a given scale is zero along with all of its descendants (as shown in Figure 4-2), then a special zerotree-root symbol (ZTR) is transmitted, eliminating the need to transmit the values of the descendants. Thus, the correlation of insignificance across scales results in a net decrease in the number of bits that must be transmitted.

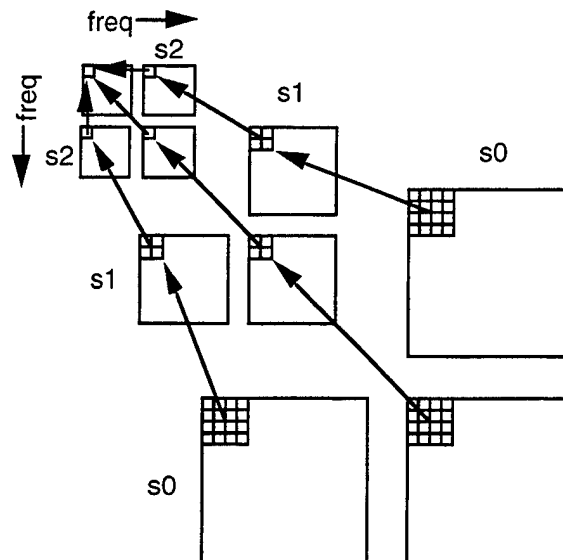


FIGURE 4-2. Wavelet Coefficient Mapping With One Complete Zerotree Shown. Note that the wavelet scale ( $s_0$ ,  $s_1$ , etc.) is inversely proportional to the spatial frequency.

In order to generate an embedded code (where information is transmitted in order of importance), Shapiro's algorithm scans the wavelet coefficients in a bit-plane fashion. Starting with a threshold determined from the magnitude of the largest coefficient, the algorithm sweeps through the coefficients, transmitting the sign (+ or -) if a coefficient's magnitude is greater than the threshold (i.e., it is significant), a ZTR if it is less than the threshold, and the root of a zerotree at the coarsest possible scale (or a 0 otherwise: this is the dominant pass). Next, for the subordinate pass, all coefficients deemed significant in the dominant pass are added to a second subordinate list, which is itself scanned. One bit

is transmitted for each coefficient on this list during the pass, decreasing its approximation error in the decoder by approximately that amount. The threshold is then halved and the two passes are repeated, with those coefficients having been found significant previously being replaced by zeros in the dominant pass (so that they do not inhibit the formation of future zerotrees). The symbol stream created by this scanning process is then passed through an arithmetic coder, to eliminate any remaining statistical redundancy before transmission to the decoder. This process continues until the bit budget is exhausted; at that point, the encoder transmits a stop symbol and its operation is terminated. The image decoder, on the other hand, simply passes the incoming bit stream through its arithmetic decoder and progressively builds up the significance map and the subordinate list in exactly the same way as they were created by the encoder. Because of this precise synchronization, the resolution enhancement bits transmitted during the subordinate pass do not need any location specifiers; the decoder knows the exact transmission order of these bits because it has reconstructed the same subordinate list as the encoder had at that point in the processing.

## PARALLEL WAVELET TRANSFORMS

### OVERVIEW

It is a well known fact that the total number of operations required to implement an algorithm in parallel is almost always larger than the number of operations required to implement it on a single processor. Thus, the optimal choice might be to simply pipeline the image compression (decompression) process by allocating one PE to each image frame as it comes in. For this to be a viable solution, the memory space controlled by each PE must be quite large. Specifically, if the original image has 8-bit pixels, then the EZW algorithm requires that 16 bits of memory be available for each image pixel (taking into account the dynamic range expansion of the transformation) plus an additional 3 bits per pixel for overhead information. Unfortunately, all of this memory must be available for the duration of the encoding/decoding process, because the embedded encoder/decoder must make numerous passes through the transformed coefficients. Pipelining also introduces at least one frame of delay for each PE in the pipeline. If one applied a large number of PEs to the coding task, a significant amount of throughput latency would be introduced into the end-to-end codec. The final problem with a pipelined encoder is input/output (I/O) collision. For the distributed memory MIMD system shown in Figure 4-1, this amounts to I/O contention along the interconnecting lines as multiple PEs try to access full images or to output coded coefficients at the same time. This is even a problem using shared memory MIMD systems having relatively few PEs. For example, in one set of experiments using the Texas Instruments (TI) 320C80 (four-integer digital signal processors (DSPs) with shared memory) processor, a speed-up was achieved by going from one pipelined processor to two, but no further speed-ups were achieved when three or four processors were used. Memory contention was found to

be the bottleneck. Because of these many drawbacks, we focus here on parallel wavelet transforms that can be implemented using an overlap-save architecture. These do not add structural latency (as opposed to computational latency) to the system, and they are well suited to a distributed memory architecture. Specifically, the image pixels can be regionally partitioned between PEs to minimize the communications cost incurred during the wavelet transformation and to eliminate the need for any communications during the encoding/decoding process.

### PERFORMANCE ANALYSIS: OVERLAP-SAVE IMPLEMENTATIONS

A number of parallel implementations of the wavelet transform have been recently studied (References 4-7 through 4-11). In all but Reference 4-7, however, the parallel implementation has been developed without carefully considering the actual application (in this case, image compression). Unlike the blocked DCT transform, all wavelet transforms (except the very simple Haar transform) use basis functions with overlapping spatial support, complicating their implementation in parallel. One straightforward way of dealing with these overlapping basis functions is to use an overlap-save methodology, in which more samples are input to the regional wavelet transform operating in each PE than are output from it. Such overlapping can be performed either once in its entirety before starting the transform, so that the transform can be iterated to its maximum depth as shown in Figure 4-3 (Reference 4-7), or progressively as shown in Figure 4-4 (References 4-9 through 4-11).

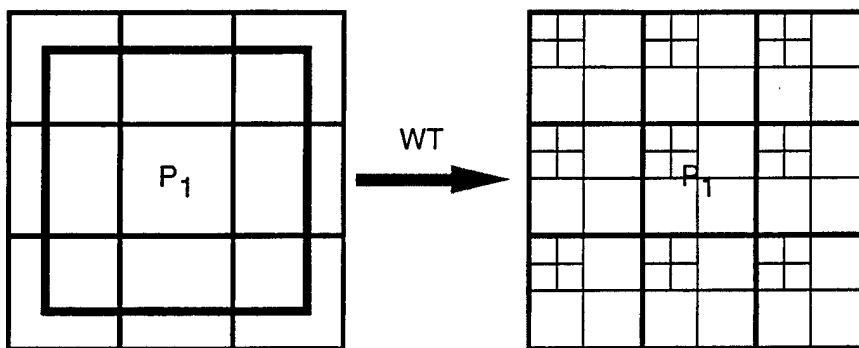


FIGURE 4-3. Wavelet Decomposition Overlap Save for Entire Image. First perform all overlap, then decompose the image in each PE (no inter-processor communications are required).

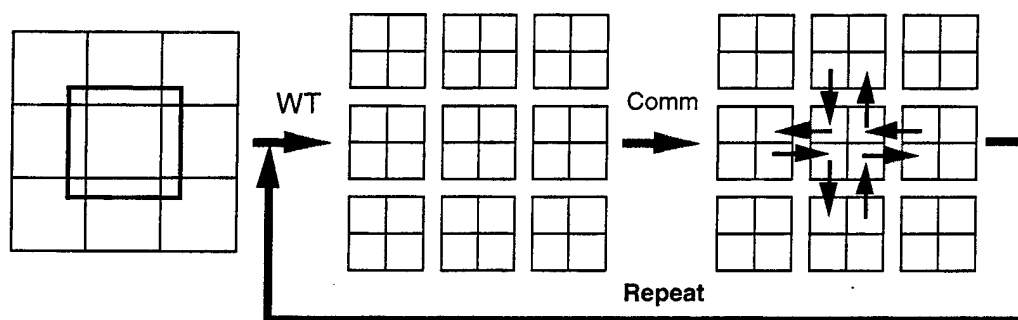


FIGURE 4-4. Wavelet Decomposition Using Overlap-Save at Each Scale. This requires inter-processor communication before the next level is computed.

In the case of Figure 4-3, all of the overlap required for a wavelet decomposition down to the coarsest scale is performed at the beginning and no communication is required during processing, while the system of Figure 4-4 requires that coefficients be exchanged after each level of the wavelet decomposition is complete. The relative trade-off between these two methods depends on the cost of implementing arithmetic operations versus the cost of moving data around. If a wavelet decomposition having even length low and highpass filters,  $N_l$  and  $N_h$ , is used then the number of samples required from the left-hand ( $O_\theta$ ) and the right-hand ( $\hat{O}_\theta$ ) PE (i.e., the PEs on either side of a specified PE in Figure 4-1) for each line or column of the image is

$$O_\theta = \hat{O}_\theta = \frac{N_\theta}{2} - 1 \quad (4-1)$$

where  $\theta$  is either l (lowpass) or h (highpass). If the lengths of the filters are odd, then the overlaps with left-hand and right-hand PEs along a scan line are given by

$$O_l = \frac{N_l - 1}{2}, \quad O_h = \frac{N_h - 3}{2}, \quad \hat{O}_l = \frac{N_l - 3}{2}, \quad \hat{O}_h = \frac{N_h - 1}{2} \quad (4-2)$$

with the difference between lowpass and highpass filters being caused by the staggered decimation required for symmetric extension at the borders (Reference 4-12). Depending on the lengths of the filters involved and the level of the wavelet decomposition, some of these samples may be drawn from non-adjacent PEs, increasing the transmission cost. At level  $k$  of the decomposition, the number of PEs completely spanned (i.e., every low frequency coefficient in that PE must be transmitted to an adjacent PE) by the overlap regions left and right of a specified PE is given by

$$P_{k,\theta} = \left\lfloor \frac{2^k O_\theta}{S} \right\rfloor \quad (4-3)$$



and

$$\hat{P}_{k,\theta} = \left\lfloor \frac{2^k \hat{O}_\theta}{S} \right\rfloor, \quad (4-4)$$

respectively, where  $\lfloor \bullet \rfloor$  is the largest integer less than the argument and  $S$  is the size of the 1D block of pixels initially contained within the PE. If each PE is treated independently, then the transmission cost incurred by one processor at level  $k$  of the wavelet decomposition in processing a single row (or column) of the image is then given by

$$C_{k,\theta} = t_k \left( 2^{-k} S \left[ 2^{P_{k,\theta}} (1 - P_{k,\theta}) + 2^{\hat{P}_{k,\theta}} (1 - \hat{P}_{k,\theta}) - 2 \right] + 2^{P_{k,\theta}} O_\theta + 2^{\hat{P}_{k,\theta}} \hat{O}_\theta \right) \quad (4-5)$$

where

$$t_k = 8 \cdot 2^k \quad (4-6)$$

is the cost in bits of transmitting one wavelet coefficient between PEs at level  $k$  (the increase in cost is due to the dynamic range expansion of the wavelet coefficients at coarser scales). Equation 4-5 is applicable when the PEs do not have enough memory available to temporarily store all of the wavelet coefficients received in one data transfer cycle. If sufficient memory is available, however, then the communications cost in Equation 4-5 is dramatically reduced to

$$C_{k,\theta} = t_k (O_\theta + \hat{O}_\theta) \quad (4-7)$$

with the assumption that the mesh is circularly connected and that circular convolution is used to cancel edge transients. To see this, consider the 1D example illustrated by Figure 4-5 in which each PE initially contains two coefficients and where the filter's left-hand overlap,  $O_l$ , equals 4. Examining the figure, we note that only four right-hand shifts of coefficients are required to accommodate the filter's left-hand overlap; similarly, four left-hand shifts can take care of the filter's right-hand overlap. Thus, Equation 4-7 is clearly true in this case. The total communications cost per PE for a depth- $L$  2-D wavelet transform using partial overlapping is therefore

$$CPO_L = 2S \cdot \left( \sum_{k=0}^{L-1} \max(C_{k,l}, C_{k,h}) \right) \quad (4-8)$$

where each PE is assumed to start with an  $S \times S$  block of image pixels. Using the fully overlapped method, added communications costs are incurred when the data are initially loaded into the parallel array. If the architecture supports direct memory access (DMA) from each PE to the image digitizer, this kind of communications can be considerably

faster than the inter-processor links of the mesh. Also, in some cases, the PEs may have *no* direct links, which would greatly decrease the efficiency of transferring data between them. The communications overhead (in bits) required to load the extra samples into each PE of the array is given by

$$\text{CFO}_L = 16S(\max(O_l, O_h) + \max(\hat{O}_l, \hat{O}_h)) \cdot (2^{L+1} - L - 1). \quad (4-9)$$

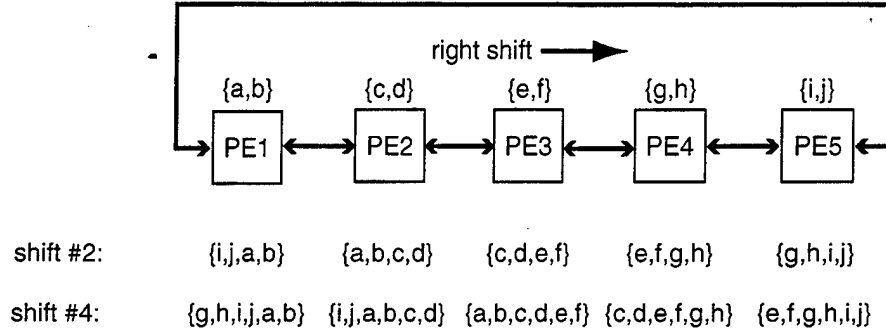


FIGURE 4-5. Example of Efficient Coefficient Transmission Using Partial Overlap Method.

The partial overlap method also increases the computational cost of implementing a wavelet transform relative to that incurred using a sequential algorithm. Specifically, the cost is increased by

$$\text{OPO}_L = 2S \sum_{i=0}^{L-1} \left\{ \frac{c_l}{2} (O_l + \hat{O}_l) + \frac{c_h}{2} (O_h + \hat{O}_h) \right\} \quad (4-10)$$

where  $c_l$  and  $c_h$  are the cost of producing one low and one high frequency coefficient, respectively. For conventional direct form implementations of these filters (i.e., not taking advantage of any filter coefficient relationships),  $c_l = 2 \cdot N_l - 1$  and  $c_h = 2 \cdot N_h - 1$  operations (additions and multiplications). We do not consider multiplications and additions separately here because most modern signal processing chips can process both with equal speed. One can also calculate the added computational costs associated with the fully overlapped method. Specifically, for a 1D block of pixels, the number of additional pixels that must be processed at scale  $k$  of the wavelet decomposition is

$$A_{k,\theta} = \frac{O_\theta + \hat{O}_\theta}{2} (2^{L-k} - 1) \quad (4-11)$$

which when summed over both filters, all scales, and two dimensions, results in a total additional cost of

$$\text{OFO}_L = 2S \sum_{k=0}^{L-1} (c_l A_{k,l} + c_h A_{k,h}). \quad (4-12)$$

This computational increase is generally larger than that of the partial overlap method given by Equation 4-9. To graphically compare the two methods, we form the equality

$$\text{OFO}_L + \alpha_1 \text{CFO}_L = \text{OPO}_L + \alpha_2 \text{CPO}_L \quad (4-13)$$

where  $\alpha_1$  and  $\alpha_2$  weight the relative cost of communications versus computation in the two cases (i.e., a large value implies that the communications cost has more impact on total performance than the computational costs). To make comparisons, we solve Equation 4-12 for  $\alpha_1$ , i.e.,

$$\alpha_1 = \alpha_2 \cdot \frac{\text{CPO}_L}{\text{CFO}_L} + \frac{\text{OPO}_L}{\text{CFO}_L} - \frac{\text{OFO}_L}{\text{CFO}_L}, \quad (4-14)$$

and plot it for different values of  $L$  (the maximum level of the wavelet decomposition) as shown in Figure 4-6, using the 5/3 biorthogonal wavelet #a/#s where #a and #s are the lengths of the analysis and synthesis lowpass filters, respectively) (Reference 4-13). The size of the image is assumed to be 512x512, the processor array has 16x16 PEs, and Equation 4-7 is used in Equation 4-8 to compute communications partial overlapping (CPO). Studying the figure, we first note that when  $L = 1$ , the slope of the line is 45 degrees, implying that both methods of overlapping are equivalent if their communications costs are equally weighted. Clearly, this must always be true. Furthermore, as  $L$  increases the slope of the line decreases, making full overlapping optimal only when its communications costs are proportionately less important than those of partial overlapping (as determined by the slope of the line). If wavelets with less compact support (i.e., longer filters) are used, we find that the slopes of these lines decrease even more rapidly as  $L$  increases. This implies that the communications advantage required for full overlapping to be equivalent to partial overlapping becomes progressively more difficult to achieve. When longer filters are used and Equation 4-5 is substituted for Equation 4-7 in Equation 4-8 (i.e., when the processors have insufficient memory), the slopes of the lines at first decrease as  $L$  increases but then begin to gradually increase, ultimately exceeding 45 degrees. Thus, the fully overlapped method actually becomes more efficient than partial overlapping for some values of  $\alpha_1$  and  $\alpha_2$ . Finally, we should point out that any solution to Equation 4-14 that requires  $\alpha_1$  to be negative is not feasible (i.e., anything below the thick horizontal line in Figure 4-6). Such a solution is equivalent to saying that the communications requirement can actually reduce the overall cost of implementing the fully overlapped method. Partial overlapping is, therefore, always the optimal choice in the region where  $\alpha_1 < 0$ . For the results

presented in the "Parallel Implementation of the EZW Coder" section below, we use the partial overlap method (simulated on a conventional serial processor) along with the 9/7 biorthogonal wavelet transform (Reference 4-13). Note that the selection of overlap methodology is irrelevant as far as the rate-distortion performance of the image coder is concerned because both parallel transforms result in identical wavelet decompositions. We use the 9/7 wavelet because it delivers excellent rate-distortion performance in image coding applications while not suffering too severely from ringing artifacts at low bit rates (Reference 4-14).

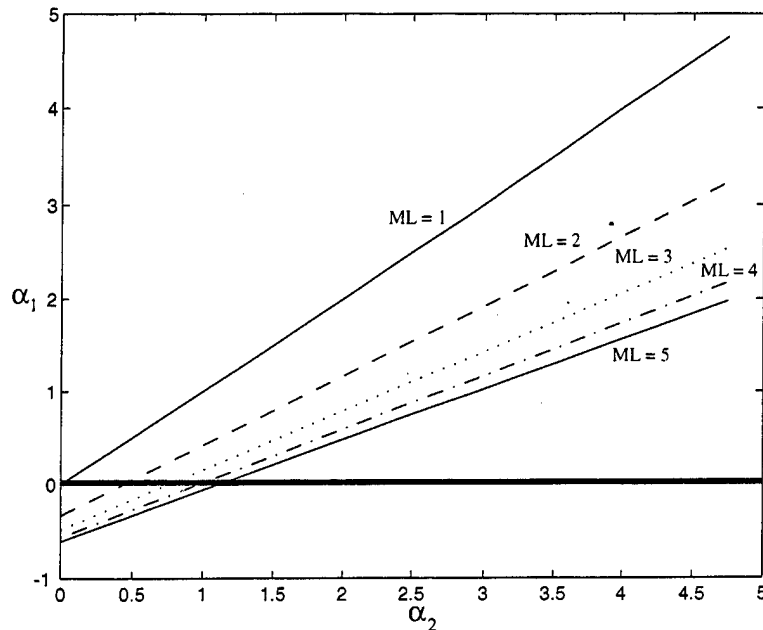


FIGURE 4-6. Comparison of Communications Costs With 5/3 Biorthogonal Wavelet (Reference 4-13) and 5 Different Levels of Decomposition (512x512 image and 256 PEs).  $\alpha_1$  is the weight on the communications cost of the fully overlapped method while  $\alpha_2$  is the weight for partial overlapping.

## APPLICATION TO IMAGE COMPRESSION

A very important question remains unanswered, however: how many PEs can one simultaneously and efficiently apply to the problem of EZW-based image compression? In References 4-9 through 4-11, the authors have tried to achieve the maximum amount of parallelism possible in their wavelet transforms, assigning one processor to each pixel in the original image. We contend here that such a strategy is wasteful for doing compression that exploits inter-scale redundancy (e.g., EZW or SPIHT (set partitioning in hierarchical trees)) because it is impossible to efficiently use all of the PEs to implement the coding and decoding portions of the compression algorithm. Why? Such coders exploit redundancy between coefficients at different scales corresponding to the same

region in the original image (see Figure 4-2) by calculating zerotrees. These calculations require that the coefficients making up a zerotree be scanned repeatedly, and this can be accomplished most efficiently if they all reside on just one processor. Another disadvantage of the 1 PE per pixel approach is that the number of processors that can be applied to the transform itself decreases by a factor of 4 for each successive level of the wavelet decomposition, leading to inefficiency (Reference 4-10). Further, as the number of PEs operating on the transform decreases, the communications costs increase because data must be consolidated into the remaining processors from all over the mesh. In short, the one PE per pixel philosophy requires more processing resources than can be effectively applied to this type of image compression algorithm. While our basic approach to implementing the overlapped transform is the same as in References 4-9 through 4-11, we do not attempt to assign one processor to each pixel of the image; rather, each processor is assigned enough pixels to ensure that it contains at least one full zerotree after the wavelet decomposition is complete. If the number of levels of wavelet decomposition is  $L$ , then the full zerotree restriction implies that the  $X_1 \times Y_1$  size subimages initially loaded into each processor are selected such that  $X_1 \geq 2^L$  and  $Y_1 \geq 2^L$ . This restriction also limits the number of parallel processors that can be easily applied to the problem. Specifically, if the original image dimensions are  $X \times Y$ , then the size of the parallel array,  $X_A \times Y_A$ , must satisfy  $X_A \leq X/X_1$ , and  $Y_A \leq Y/Y_1$ . Note that the subimage sizes  $X_1$  and  $Y_1$  must be powers of 2 to ensure that each processor includes only complete zerotrees (zero-padding can be used at the image boundaries if necessary). Clearly, the zerotree requirement significantly constrains the amount of parallelism allowed and reduces the flexibility with which processing elements can be applied (i.e., processors are optimally added in powers of 4). On the plus side, however, it ensures that all PEs remain busy during the wavelet processing and that the coding can be done in parallel with virtually no inter-processor communications.

The computational complexity of the inverse transform is the same as that of the forward transform while the amount of data movement required is essentially the same. For image decompression, a relatively small amount of data must be initially moved into each PE (the coded symbols) and the wavelet coefficients are then generated from these data (with each PE then containing complete zerotrees). Coefficients must then be traded among PEs so that each will have what it needs to invert one level of the wavelet transform in a specific region of the image. Coefficients are then traded again, with the process continuing until the complete image has been reconstructed. Thus, computing the inverse wavelet transform on an array of parallel processors is essentially the same operation as shown in Figure 4-4, but with the communications and (inverse) transform steps reversed. The amount of data that must be transmitted from one PE to another is the same as in Equation 4-5 for the forward transform, although one should note that in this case the processing is performed from coarse to fine scale.

## PARALLEL IMPLEMENTATION OF THE EZW CODER

In this section, we quantify the trade-offs associated with the direct implementation of the EZW coding algorithm on multiple, parallel processors. By direct implementation we mean that a separate but identical copy of the zerotree encoder independently processes wavelet coefficients within each of the  $P = X_A Y_A$  processors. To allocate bits among the  $P$  processors, we use the entropy of the image region contained within each one prior to the wavelet decomposition. Thus, a region with lower entropy (implying that it contains less information) receives a smaller bit allocation than one with higher entropy. Unfortunately, independently processing groups of wavelet coefficients with separate encoders reduces the statistical correlation in the scanned symbol streams, decreasing the efficiency of the arithmetic coding process. As the number of parallel processors increases in this simplistic scheme, the drop in arithmetic encoding efficiency results in reduced rate-distortion performance.

A very important consideration is how to get data into and out of the processor array. The information coming into the array (i.e., the image pixels) requires far more communications bandwidth than the data going out of the array. Therefore a real-time solution for video compression will probably not be effective without some sort of DMA from the individual PEs to the frame grabber. Most practical architectures that have been designed for video processing have such provisions (e.g., TI 320C80s and Analog Devices SHARCs). Because the data rate out of the encoder is much lower, we have more options. Because each PE is independently running its own EZW encoder on different data sets, the compressed output bits are produced in an asynchronous manner. Thus on a given clock cycle, one PE may produce a bit while another does not. There are two basic ways to solve this problem: (1) wait until every PE has produced its byte and then shift them out of the array or (2) store compressed data in the local memory of each PE and output it when the coding is complete. The first approach can significantly slow down processing and is, consequently, not preferred. If fast DMA access is available, the latter approach is much better. If all of the PEs try to output compressed data simultaneously, however, collisions can result, and the speed of the algorithm can be greatly reduced. This is where the asynchronous nature of the coding process actually becomes advantageous. The speed with which each PE finishes its coding task depends on the complexity of the image region corresponding to its wavelet coefficients: the more complex the region, the longer it takes. As more regional subdivisions are used (i.e., more processors), the variance of the processing times increases dramatically. Thus if each PE sends out its compressed bits when it finishes, it is unlikely that there will be too many other PEs also requesting data transfers. It is important to note as well that the amount of data being output is much less than the amount being input, so the system performance is primarily limited by the data path from the digitizer to the array.

Obviously, the decoder suffers from a similar but inverted problem: a relatively small amount of compressed data is initially read into each PE, and a large amount of decompressed data is read out. For the same reasons as above, the speed at which image pixels can be read out is the key to achieving good real-time performance. In this case, the input data can be read in and processed as they are received, with the caveat that all of the wavelet coefficients must be decoded before the inverse wavelet transform is performed. For all practical purposes, the inverse wavelet transformation is completed simultaneously in all PEs, so the reconstructed image pixels are all available for output at the same time. Again, an architecture that supports DMA directly from the PEs' local memory to a display device would be desirable here.

The results of this parallel coding algorithm in terms of the bit rate (bits per pixel (bpp)) versus the peak signal to noise ratio are summarized in Figure 4-7 for the 512x512 "Lena" image. For the results shown in Figure 4-7, each processor computes and transmits the mean of its block of pixels and the maximum wavelet coefficient value ("maxval") of the block after transformation. In contrast, the conventional EZW encoder computes and transmits one global mean and one global maxval. Transmitting the local mean and maxval for each processor eliminates the need for any communications between the PEs during the coding process, but it also results in a significant rate-distortion penalty as the number of PEs increases. Worse yet, removing local mean values before the wavelet transformation (instead of the global mean value) results in blocking artifacts at low bit rates (see Figure 4-8). If instead a single mean and maxval are sent for the entire image, the rate-distortion performance of the parallel implementation greatly improves, as illustrated by Figure 4-9. Computing a global mean and maxval requires approximately  $2(X_A + Y_A)$  communications operations per PE in the encoder, but this is a small price to pay for such a dramatic improvement in performance. For example, if 256 processors are used, the PSNR shown in Figure 4-9 is about 2 decibels higher than that of Figure 4-7 at bit rates between 0.07 and 0.26 bpp. Figures 4-10 through 4-14 visually illustrate some of these parallel trade-offs for the Lena image coded at 0.25 bpp.

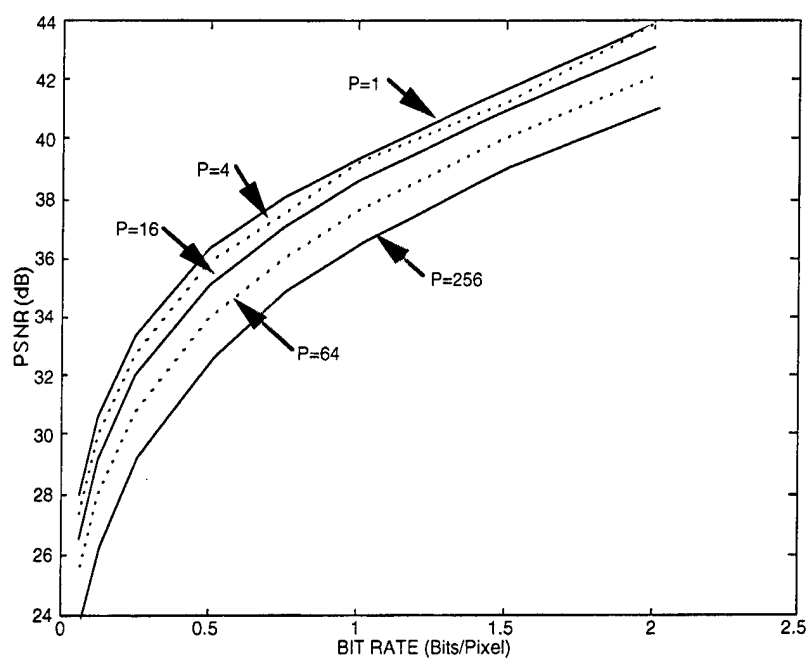


FIGURE 4-7. EZW Coding Results for the 512x512 Lena Image With a Separately Computed Mean and maxval for Each Processor. Each of  $P$  processor operates on square blocks of image pixels with alternating solid and dotted lines included for clarity.



FIGURE 4-8. 0.25 bpp,  $P = 256$ , Separate Means and maxvals, PSNR = 29.62 dB.



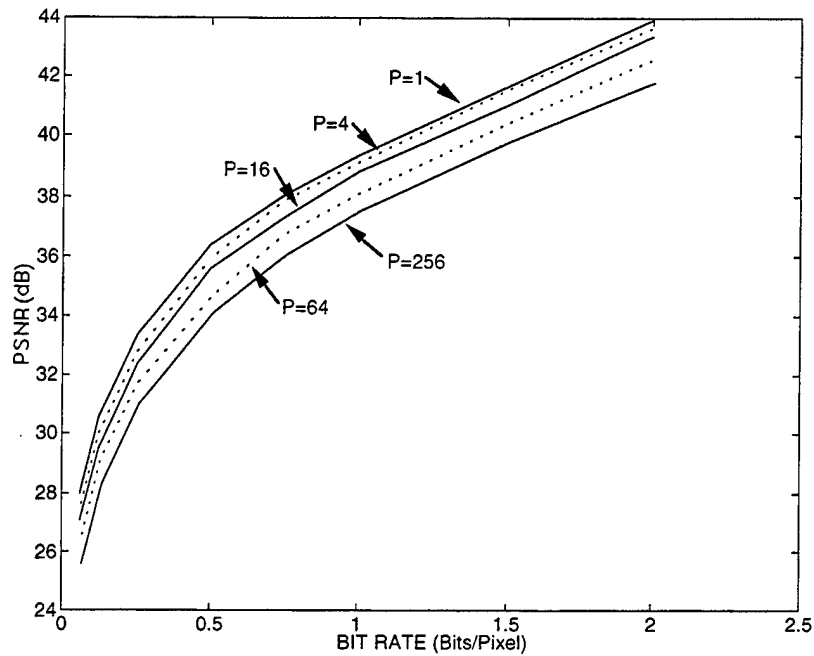


FIGURE 4-9. EZW Coding Results for the 512x512 Lena Image With Common Mean and maxval for All Processors. Each of P processor operates on square blocks of image pixels with alternating solid lines and dotted included for clarity.



FIGURE 4-10. Original Lena Image, 512x512 Pixels.



FIGURE 4-11. 0.25 bpp, Conventional EZW, PSNR = 33.37 dB.



FIGURE 4-12. 0.25 bpp,  $P = 4$ , Separate Means and maxvals, PSNR = 32.77 dB.



FIGURE 4-13. 0.25 bpp,  $P = 4$ , Global Mean and maxval, PSNR = 32.80 dB.



FIGURE 4-14. 0.25 bpp,  $P = 256$ , Global Mean and maxval, PSNR = 31.32 dB.

When studying the figures, we note that the rate-distortion penalty incurred by using a parallel implementation seems to be directly proportional to the number of PEs (a number that increases exponentially in the plots). Thus, the penalty for using just four processors in parallel is quite acceptable while that for using 256 processors may not be (-2 decibels lower PSNR than in the single processor case at all bit rates in Figure 4-9). We should also point out other features of a direct parallel implementation of the EZW algorithm. First, the embedded structure of the algorithm can be maintained by interleaving the bits output from the PEs before transmitting them. Because the data rate is relatively low (it operates on compressed data), interleaving can be performed with a relatively slow processor having only a modest amount of memory. It will, however, add latency to the system, so it may not be suitable for applications requiring very low throughput delay. In addition, because different regions of the image are coded independently by each PE, the parallel-structured coder is more resistant to transmission errors: if a single error occurs, only one section of the image is impacted (it will be reconstructed at lower resolution than the others) (Reference 4-15). In a conventional EZW image coder, on the other hand, one bit error reduces the quality of the entire reconstructed image. Finally, we note that a stop symbol is normally transmitted by each processor to terminate its bit stream. If the bit allocation is determined *a priori*, this added symbol can be completely eliminated, improving the rate-distortion performance slightly at lower bit rates and with higher levels of parallelism.

Whether global or local mean and maxval quantities are transmitted, the complexity of the parallel zerotree decoder is the same. In either case, the appropriate mean and maxval are first broadcast to each processor and then the bit stream is parsed and properly distributed among the processors as it arrives. The decoders operating in each PE are identical to the sequential EZW decoder of Reference 4-1 and are completely independent of each other, making the overall parallel implementation very efficient and highly scalable.

## MODIFIED PARALLEL IMPLEMENTATION

As noted above, a significant performance penalty is incurred when the EZW algorithm is directly mapped to multiple processors. We propose here an architecture that eliminates this penalty at the expense of added computational complexity. Figure 4-15 illustrates the basic idea. The processors in the mesh perform the transform, calculate the zerotrees, and output the symbols (+, -, 0, and ZTR (zerotree root) for the dominant list) to a single processor, which codes them using a small-alphabet arithmetic encoder. It turns out that all of these processes can be performed using synchronous SIMD PEs and, in fact, that there is no real advantage to even considering MIMD devices because rigid output synchronization must be maintained. Other authors have shown that the wavelet transform can be implemented on an SIMD array (Reference 4-9), but we have determined that the zerotree formation and symbol generation processes can also be

performed synchronously. The details of this implementation are beyond the scope of this paper, but suffice it to say that each PE executes exactly the same instruction at the same time on different data: our SIMD code contains no "if" statements whatsoever. To maintain this structure, a concession must be made. Each PE would normally generate output symbols in an asynchronous way because of the compression inherent in the ZTR symbol and the fact that differing numbers of significant symbols (i.e., + and -) are generated. However to maintain synchronization between the PEs and the output processor, each PE must output its symbol at a predetermined time. If a given PE does not have a true code symbol to send, then a dummy symbol is used instead, which the output processor knows to ignore. The amount of communications required between the array and the output processor is clearly somewhat high, but because only one symbol (2 or 3 bits) is sent for every 20 instructions executed, it should be feasible with many architectures (i.e., those which can execute a communications and an arithmetic operation simultaneously) to use a bucket-brigade style pipeline to keep the symbols flowing toward the output processor. In fact if this approach is used, no symbol reordering is needed prior to arithmetic encoding at the output processor. In addition, by using a common mean and maxval for all processors, the parallel architecture described by Figure 4-15 can achieve performance identical to that of the conventional zerotree algorithm. In order to achieve a fixed bit rate output stream, the output processor must also monitor its bit production and turn off the mesh processors when the desired bit rate is achieved. Finally, one should note that while the output processor does have to run fairly fast, it does not have to run  $P$ -times as fast (where  $P$  is again the number of parallel processors) because each PE executes approximately 20 instructions for each symbol it creates, and most of the symbols are discarded without arithmetic encoding.

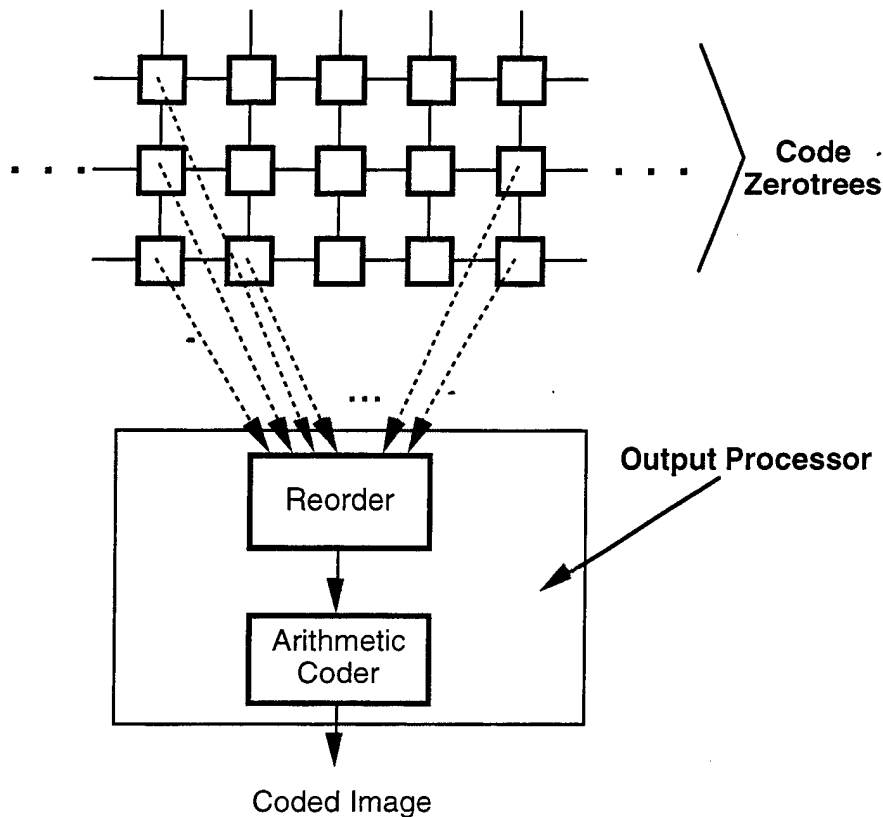


FIGURE 4-15. Modified Parallel EZW Image Coding Architecture.

A parallel decoder for this modified scheme has essentially the same architecture as the encoder (i.e., an input processor that performs the arithmetic decoding and an array of mesh processors that successively approximate the wavelet coefficients). Unfortunately, in this case more processing must be performed in the input processor. Specifically, the incoming bits for a given pass (i.e., the symbols generated by processing 1 bit plane of the wavelet coefficients) are first arithmetically decoded and the ZTR symbols converted into appropriate blocks of zero-valued coefficients. These symbols must then be transferred to the processor array, where non-zero coefficients are approximated and where, eventually, the inverse wavelet transformation is performed. While the parallel encoder operates in a more or less flow-through fashion, the decoder must operate in what is basically a batch mode. Furthermore, it requires a more powerful input processor relative to the power of its PE array when compared to the encoder, because it must also decode the ZTR symbols. One should note, however, that an EZW decoder is always less complex than its corresponding encoder, so it would still probably be possible to implement them both using the same single/multiple processor hardware configuration.

## PERFORMANCE COMPARISONS ON THE TI 320C80

Thus far, we have only presented performance comparisons in which parallel operation had been simulated on a sequential processor. While such analysis provides valuable insight into some of the design trade-offs that are possible within a parallel framework, it is not completely satisfactory. Ultimately, one would like to actually run the algorithm in parallel and find out how it really works. As part of an ongoing project, we have implemented wavelet compression and decompression algorithms on a TI 320C80 multivideo processor. This chip contains 5 MIMD parallel processors, 4 integer and 1 floating point, with cache-controlled global memory, and it is capable of performing in excess of 1.8 billion operations per second. To perform image analysis in the encoder, we use a 5/3 biorthogonal wavelet transform operating on the four integer PEs, with partially overlapped parallel partitioning. The video frames fed into our encoder are 512x240 pixels, and we decompose each to 4 levels vertically and 5 horizontally. Next, the wavelet coefficients created by this decomposition are encoded separately by each of the 4 integer PEs as described above. The encoder used here is very similar to the one summarized in the "Imaging Platform" section of this report, except that it contains some enhancements that provide additional speedups (References 4-16 and 4-17). Considering execution times for the wavelet transform alone, running on all 4 of the PEs in parallel is 3.22 times faster than running on just 1 sequentially (16 versus 52 milliseconds for a single frame of video). This is about what we would expect because the overlap inherent in the wavelet transform requires each PE to process and communicate extra data. When just *coding* the coefficients, however, the 4-processor system is 3.96 times faster (90 versus 356 milliseconds): almost exactly a linear speedup, as expected! Overall, we achieve a frame rate of 9.4 frames per second (fps) using the parallel system versus 2.4 fps for an identical sequential system. In both cases, the rate-distortion performance is exactly the same because it is limited by the cache-based zerotree processing requirement imposed by Reference 4-18.

## PARALLEL COMPRESSION USING SPATIAL ORIENTATION TREES

An image compression algorithm was recently introduced by Said and Pearlman that also exploits inter-scale correlation to achieve excellent rate-distortion performance (Reference 4-2), better in fact, than the original EZW algorithm. This new algorithm organizes the wavelet coefficients into spatial orientation trees that are then used to form coefficient sets and searched (to determine a coefficient's significance) in a way similar to the EZW coder. As with EZW, this coder can be parallelized efficiently with no or

minimal communications if a complete spatial orientation tree is contained within each PE. To ensure this, the image subdivision must be formed so that every PE contains 4 complete zerotrees. If the number of levels of wavelet decomposition is again  $L$ , this constraint implies that the size of the subimage in each processor prior to the wavelet decomposition,  $X_1 \times Y_1$ , must be at most  $X_1 = 2^{L-1}$  and  $Y_1 = 2^{L-1}$ , which in turn limits the size of the parallel array to  $X_A = X/X_1$ , and  $Y_A = Y/Y_1$  (for an  $X \times Y$  image). Thus, using this coding algorithm in place of the conventional EZW algorithm reduces the allowable parallelism by a factor of 4. Because of their more sophisticated arithmetic coder, it is also likely that an increased penalty might be paid for higher levels of parallelism when using the direct implementation discussed above. The modified parallel architecture discussed here should also work with this algorithm and would not significantly reduce the rate-distortion performance if properly implemented.

## CONCLUSIONS

In this work, we have considered the complete problem of implementing a wavelet-based image compression algorithm on an array of processors operating in parallel. In particular, we have considered both MIMD and SIMD implementations, discussing their strengths and weaknesses. As part of this process, we have studied parallel implementations of the wavelet transform and have postulated the "ideal" level of parallelism for this application (i.e., enough PEs so that every one contains at least one complete zerotree after decomposition). We have also analyzed the partitioning of the coding algorithm itself, quantifying the performance penalty incurred by parallelization and introducing a way to overcome this penalty by using a distributed memory SIMD architecture. While much past research has been done on parallelizing the wavelet transform, it has seldom been applied in the context of a complete image compression application (i.e., it has not considered parallelization of the quantization and coding processes). We have done that here for an important class of high performance algorithms that exploit inter-scale redundancies. In a more general sense, we hope that our work has reinforced the view that it is important to consider the entire application, not just one part of it, when studying the problem of efficient parallel implementation.



## REFERENCES

- 4-1. J. M. Shapiro. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. on Signal Processing*, Vol. 41, No. 12 (December 1993), pp. 3445-3462.
- 4-2. A. Said and W. A. Pearlman. "A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," *IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 6, No. 3 (June 1996), pp. 243-250.
- 4-3. Z. Xiong, K. Ramchandran, and M. T. Orchard. "Joint Optimization of Scalar and Tree-Structured Quantization of Wavelet Image Decompositions," *Proc. of the 27th Annual Asilomar Conf. on Signals, Systems, and Computers* (November 1993), pp. 891-895.
- 4-4. D. Taubman and A. Zakhor. "Multirate 3-D Subband Coding of Video," *IEEE Trans. on Image Processing*, Vol. 3, No. 5 (September 1994), pp. 572-588.
- 4-5. C. D. Creusere. "Embedded Zerotree Image Coding Using Low Complexity IIR Filter banks," *Proc. Int. Conf. on Acoustics, Speech, and Signal Proc.* (May 1995), pp. 2213-2216.
- 4-6. C. D. Creusere. "Image Coding Using Parallel Implementations of the Embedded Zerotree Wavelet Algorithm," *Proc. of the SPIE (Digital Video Compression: Algorithms and Technologies 1996)*, Vol. 2668 (February 1996), pp. 82-92.
- 4-7. K. H. Goh, J. J. Soroghan, and T. S. Durrani. "Parallel 3-D Wavelet Transform Codec for Image Sequences," *Transputer Applications and Systems '93: Proc. of the World Transputer Congress* (September 1993), pp. 701-711.
- 4-8. C. Chakrabarti and M. Vishwanath. "Efficient Realizations of Discrete and Continuous Wavelet Transforms: From Single Chip Implementations to Mapping on SIMD Array Computers," *IEEE Trans. on Signal Proc.*, Vol. 43, No. 3 (March 1995), pp. 759-771.
- 4-9. J. Lu. "Parallelizing Mallat Algorithm for 2-D Wavelet Transforms," *Information Proc. Letters*, Vol. 45 (1993), pp. 255-259.
- 4-10. J. Lu. "Computation of 2-D Wavelet Transform on the Massively Parallel Computer for Image Processing," *Proc. of the 7th IEEE Multi-D Signal Proc. Workshop* (September 1991).

- 4-11. H. J. Lee, J. C. Liu, A. K. Chan, and C. K. Chui. "Parallel Implementation of Wavelet Decomposition/Reconstruction Algorithms," *Proc. of the SPIE (Wavelet Applications)*, Vol. 2242 (1994), pp. 248-259.
- 4-12. E. H. Adelson, E. Simoncelli, and R. Hingorani. "Orthogonal Pyramid Transforms for Image Coding," *Proc. of the SPIE*, Vol. 845 (October 1987), pp. 50-58.
- 4-13. I. Daubechies. *Ten Lectures on Wavelets*, Philadelphia, PA, SIAM, 1992.
- 4-14. M. Lightstone and E. Majani. "Low Bit-Rate Design Considerations for Wavelet-Based Image Coding," *Proc. of the SPIE Symposium on Visual Communications and Image Processing*, Vol. 2308, Pt. 1 (September 1994), pp. 501-512.
- 4-15. C. D. Creusere. "A New Method of Robust Image Compression Based on the Embedded Zerotree Wavelet Algorithm," *IEEE Trans. on Image Processing*, Vol. 6, No. 10 (October 1997), pp. 1436-1442.
- 4-16. C. D. Creusere. "Adaptive Embedding for Reduced Complexity Image and Video Compression," *Proc. of the SPIE*, Vol. 3309, pp. 48-57.
- 4-17. C. D. Creusere. "Fast Embedded Video Compression Using Cache-Based Processing," *Proc. of the European Signal Processing Conf.* (September 1998), Isle of Rhodes, Greece.

## CHAPTER 5.

### ROBUSTNESS TO TRANSMISSION ERRORS

#### SUMMARY

In this work, we present a new family of image compression algorithms derived from Shapiro's EZW coder. These algorithms introduce robustness to transmission errors into the bit stream, while still preserving its embedded structure. This is done by partitioning the wavelet coefficients into groups, coding each group independently, and interleaving the coded bit streams for transmission; thus if one bit is corrupted, then only one of these bit streams will be truncated in the decoder. To evaluate these algorithms, we have analyzed them in a stochastic framework. By combining the results of this analysis with the actual robust EZW coder, we can objectively and subjectively compare the expected reconstructed images generated by the new algorithms to those output by the original. These comparisons clearly show that all of the members of the robust family are superior to the conventional algorithm in realistic transmission environments; further, they facilitate the selection of the optimal member of this family for a given channel. Finally, we note that the new algorithms do not increase the complexity of the overall system and, in fact, that they are far more easily parallelized than the conventional EZW coder.

#### INTRODUCTION

Recently, the proliferation of wireless services and the Internet along with the consumer demand for multimedia products have spurred interest in the transmission of image and video data over noisy communications channels whose capacities vary with time. In such applications, it can be advantageous to combine the source and channel coding (i.e., compression and error correction) processes from both a complexity and an information theory standpoint (Reference 5-1). In this chapter, we introduce a form of low complexity joint source-channel coding in which varying amounts of transmission error robustness can be built directly into an embedded bit stream. The approach taken here modifies Shapiro's EZW image compression algorithm (Reference 5-2), but the basic idea can be easily applied to other wavelet-based embedded coders such as those of Said and Pearlman (Reference 5-3) and Taubman and Zakhor (Reference 5-4). Some preliminary results using this approach have previously been presented (see References 5-5 and 5-6).

This chapter is organized as follows: we first discuss the conventional EZW image compression algorithm and its resistance to transmission errors. Then we develop our new, robust coder and explore the options associated with its implementation. We then analyze the performance of the robust algorithm in the presence of channel errors and we use the results of this analysis to perform comparisons. Finally, we discuss implementation and complexity issues, followed by our conclusions.

## EZW IMAGE COMPRESSION

After performing a wavelet transform on the input image, the EZW encoder progressively quantizes the coefficients using a form of bit plane coding to create an embedded representation of the image (i.e., a representation in which a high resolution image also contains all coarser resolutions). This bit plane coding is accomplished by comparing the magnitudes of the wavelet coefficients to a threshold  $T$  to determine which of them are significant: if the magnitude is greater than  $T$ , that coefficient is significant. As the scanning progresses from low to high spatial frequencies, a 2-bit symbol is used to encode the sign and position of all significant coefficients. This symbol can be a + or -, indicating the sign of the significant coefficient; a 0 indicating that the coefficient is insignificant; or a ZTR indicating that the coefficient is insignificant along with all of the finer resolution coefficients corresponding to the same spatial region. The inclusion of the ZTR symbol greatly increases the coding efficiency because it allows the encoder to exploit inter-scale correlations that have been observed in most images (Reference 5-2). After computing the "significance map" symbols for a given bit plane, resolution enhancement bits must be transmitted for all significant coefficients; in our implementation, we concatenate two of these to form a symbol. Prior to transmission, the significance and resolution enhancement symbols are arithmetically encoded using the simple adaptive model described in Reference 5-7, with a 4-symbol alphabet (plus 1 stop symbol). The threshold  $T$  is then divided by 2 and the scanning process repeated until some rate or distortion target is met. At this point, the stop symbol is transmitted. The decoder, on the other hand, simply accepts the bit stream coming from the encoder, arithmetically decodes it, and progressively builds up the significance map and enhancement list in the exact same way as they were created by the encoder.

The embedded nature of the bit stream produced by this encoder provides a certain degree of error protection. Specifically, all of the information that arrives before the first bit error occurs can be used to reconstruct the image; everything that arrives after is lost. This is in direct contrast to many compression algorithms where a single error can irreparably damage the image. Furthermore, we have found that the EZW algorithm can actually detect an error when its arithmetic decoder terminates (by decoding a stop symbol) before reaching its target rate or distortion. It is easy to see why this must happen. Consider that the encoder and decoder use the same backward adaptive model to calculate the probabilities of the 5 possible symbols (4 data symbols plus the stop

symbol) and that these probabilities directly define the codewords. Not surprisingly, the length of a symbol's codeword is inversely proportional to its probability. If a completely random bit sequence is fed into the arithmetic decoder, then the probability of decoding any symbol is completely determined by the initial state of the adaptive model (i.e., the probability weighting defined by the model is not, on the average, changed by a random input).

In our implementation of the Witten et al arithmetic coder of Reference 5-7, we set Max\_frequency equal to 500 and maintain the stop symbol probability at  $1/\text{cum\_freq}[0]$ . Because cum\_freq[0] (the sum of the frequency counts of all symbols) is divided by 2 whenever it exceeds Max\_frequency, the probability of decoding a stop symbol stays mostly between 1/250 and 1/500. Thus, if a random bit stream is fed into the decoder after training it to this point, an average of 250 to 500 symbols will be processed before the stop symbol is decoded. The bit stream is correctly interpreted as long as the decoder is synchronized with the encoder, but this synchronization is lost shortly after the first error occurs. Once this happens, the incoming bit stream looks random to the decoder (the more efficient the encoder, the more random it will appear). Because each symbol is represented in the compressed image by between one and two bits, the decoder should self-terminate between 31 and 125 bytes after an error occurs. Experimentally, we have found that the arithmetic decoder overrun is typically between 30 and 50 bytes, which is consistent with the theoretical range because most of these terminations took place while decoding the highly compressed significance map. If the overrun is small compared to the number of bits correctly decoded, it does not significantly affect the quality of the reconstructed image. While some erroneous information is incorporated into the wavelet coefficients, the bit plane scanning structure ensures that it is widely dispersed spatially, making it visually insignificant in the image.

## REZW ALGORITHM

### BASIC APPROACH

As shown in Figure 5-1, the basic idea of the REZW image compression algorithm is to divide the wavelet coefficients up into S groups and then to quantize and code each of them independently so that S different embedded bit streams are created. These bit streams are then interleaved as appropriate (e.g., bits, bytes, packets, etc.) prior to transmission so that the embedded nature of the composite bit stream is maintained. In the remainder of this chapter, we assume that individual bits are interleaved. For the REZW approach to be effective, each group of wavelet coefficients must be of equal size and must uniformly span the image. A similar method has been proposed in Reference 5-8 to parallelize the EZW algorithm, but that method instead groups the coefficients so that data transmission between processors is minimized.

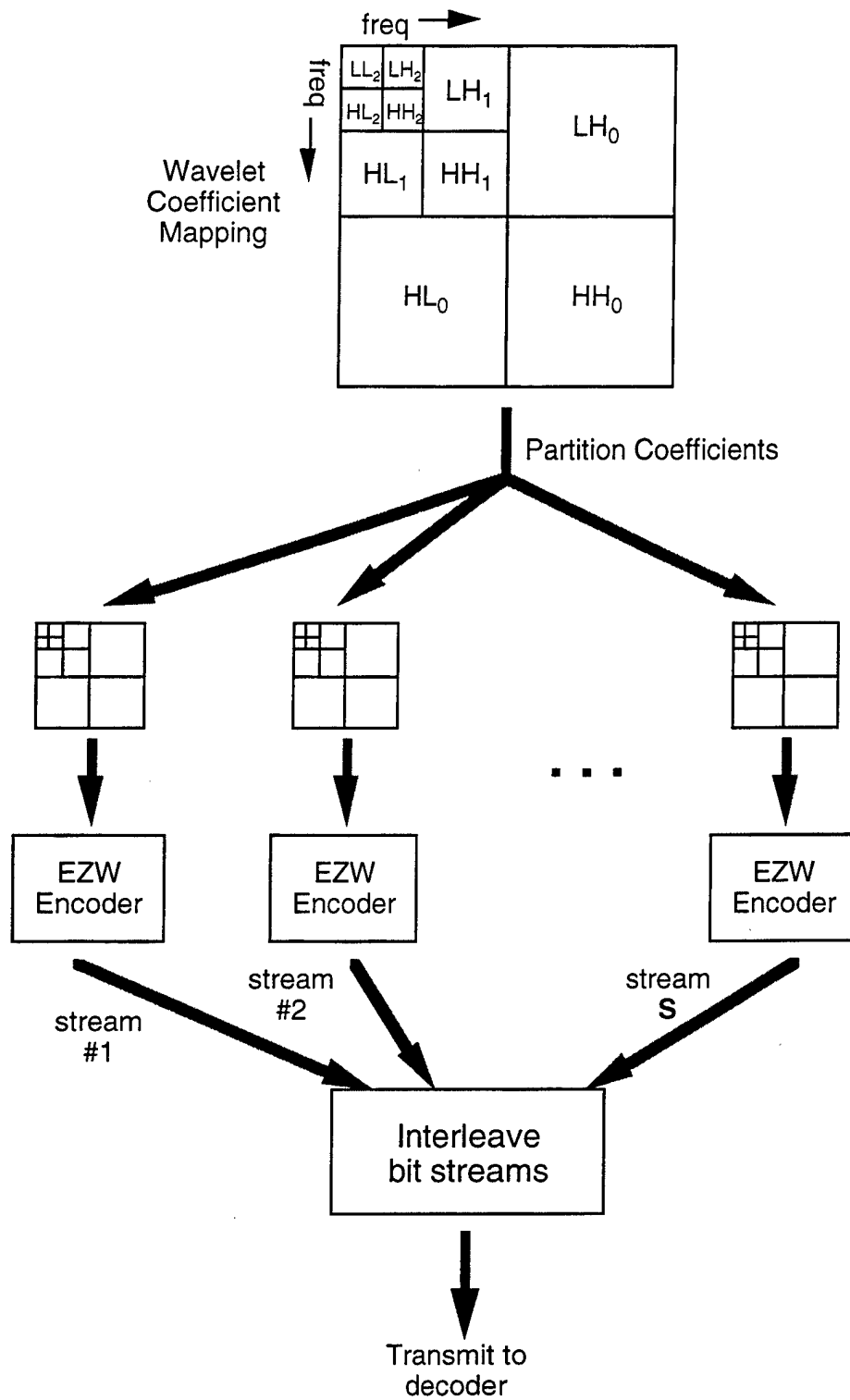


FIGURE 5-1. Structure of the REZW Algorithm.

What do we gain by using this new algorithm over the conventional one? As pointed out above, the EZW decoder can use all of the bits received before the occurrence of the first error to reconstruct the image. By coding the wavelet coefficients with multiple, independent (and interleaved) bit streams, a single bit error truncates only one of the streams; the others are still completely received. Consequently, the wavelet coefficients represented by the truncated stream are reconstructed at reduced resolution, while those represented by the other streams are reconstructed at the full encoder resolution. If the set of coefficients in each stream spans the entire image, then the inverse wavelet transform in the decoder evenly blends the different resolutions so that the resulting image has a spatially consistent quality.

### ZEROTREE PRESERVING (ZP) PARTITIONING

Figure 5-2 graphically illustrates this wavelet coefficient partitioning for  $S = 4$  bit streams and three wavelet scales. In the figure, each coefficient with the same shade of gray maps to the same group and is, therefore, processed by the same encoder. Furthermore, boxes with Xs in them are used to define the elements of one zerotree, and we note that this zerotree is identical to those used by the conventional EZW coder (Reference 5-2); hence the name "zerotree preserving." It is clear from this example that all of the elements from a given zerotree are fed into the same encoder and thus, that the correlation of insignificant coefficients between scales is fully exploited. Note that  $S$  can be increased by powers of 4 until the encoder in each stream processes just one zerotree. If the image is of size  $X \times Y$  (assuming for simplicity that both of these are powers of 2) and maximum number of scales ( $NS$ ) of wavelet decomposition are used, then the single zerotree limitation implies that the maximum number of independent bit streams allowed is  $S = (X \cdot Y) / 4^{NS}$ . While introducing more bit streams is advantageous in terms of robustness to errors, it also results in reduced rate-distortion performance when no transmission errors occur (see the "Results" section below). The partitioning of scale  $j$  (as labeled in Figure 5-1) of the wavelet coefficient mapping  $W_j(x, y)$  into  $S = 4^K \leq (X \cdot Y) / 4^{NS}$  groups, is given by

$$W_{\Psi, j}(x, y) = W_j \left( \begin{array}{l} 2^{NS-j-1} \left\{ \left\lfloor \frac{x}{2^{NS-j-1}} \right\rfloor \cdot (2^K - 1) + n \right\} + x, \\ 2^{NS-j-1} \left\{ \left\lfloor \frac{y}{2^{NS-j-1}} \right\rfloor \cdot (2^K - 1) + m \right\} + y \end{array} \right) \quad (5-1)$$

where  $\Psi = n + 2^K \cdot m$  specifies the stream number for  $\{n, m\} \in [0, 2^K - 1]$  and  $\lfloor \bullet \rfloor$  is the largest integer less than the argument. Note that the scale  $j$  is inversely proportional to the frequency: e.g.,  $j = 0$  and  $j = NS - 1$  reference the highest and lowest frequency subbands, respectively.

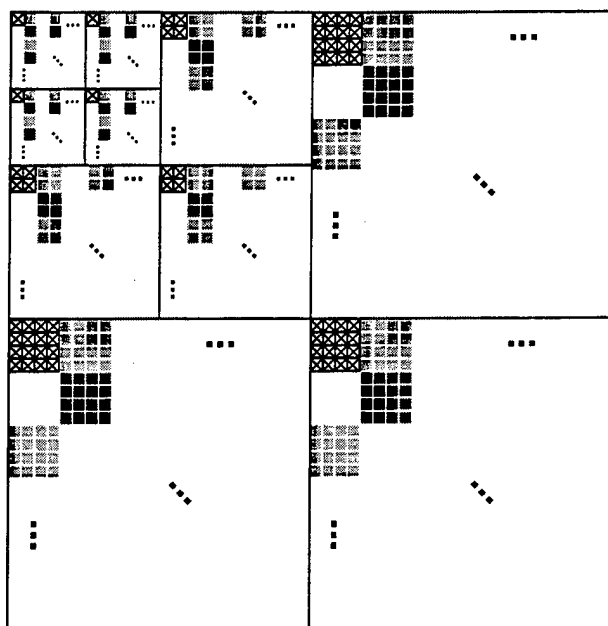


FIGURE 5-2. Zerotree Preserving Wavelet Coefficient Partitioning for  $S = 4$  and 3 Wavelet Scales. All coefficients with the same shade map to the same stream, and the Xs denote one complete zerotree.

### OFFSET ZEROTREE (OZ) PARTITIONING

A second partitioning of the wavelet coefficients is illustrated by Figure 5-3 when  $S = 4$  and  $NS = 3$ . In this case, the parent-child relationships of the conventional zerotree structure are no longer preserved (as indicated by the boxed Xs in the figure). Instead, these relationships are staggered between wavelet scales in such a way that no adjacent wavelet coefficients are input to the same encoder stream. In the more general case where  $S = 4^k \leq (X \cdot Y)/4^{NS}$ , every coefficient in a given stream is separated from any others in the same stream by at least  $2^k - 1$  wavelet coefficients. As with the zerotree preserving partitioning above, every  $2^k$ th sample (in both the horizontal and vertical directions) of the four coarsest scales is mapped to the same stream. In the offset zerotree partitioning, however, this  $2^k$ th resampling is applied at all other wavelet scales as well (i.e., Equation 5-1) is applied to all scales with  $j = NS - 1$ ). The advantage of this method over ZP partitioning is that if one bit stream is terminated prematurely, its reduced-resolution coefficients will be surrounded by full resolution coefficients at every scale. Thus, the image reconstructed by the inverse wavelet transform retains more of its high frequency information when selective transmission errors occur. Unfortunately, the zerotrees are now spatially staggered between scales, reducing the inter-scale correlation between wavelet coefficients and, consequently, the rate-distortion performance of the coding algorithm.



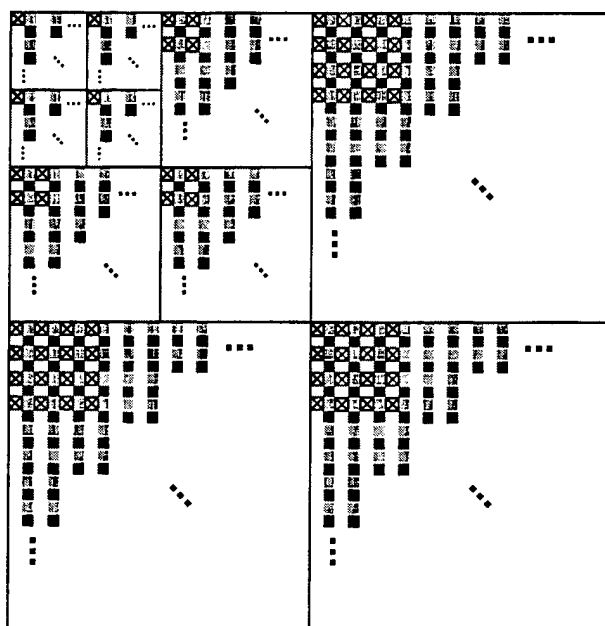


FIGURE 5-3. Offset Zerotree Wavelet Coefficient Partitioning for  $S = 4$  and 3 Wavelet Scales. All coefficients with the same shade map to the same stream, and the Xs denote one complete zerotree.

### STOCHASTIC ANALYSIS

To evaluate the effectiveness of this family of robust compression algorithms, we assume that the coded image is transmitted through a binary symmetric, memoryless channel with a probability of bit error given by  $\epsilon$ . We would like to know the number of bits correctly received in each of the  $S$  streams. Because this quantity is itself a random variable, we use its mean value to characterize the performance of the different algorithms. Because the channel is memoryless, streams terminate independently of each other, but the mean values of their termination points are always the same for a specified  $\epsilon$ . Assuming that the image is compressed to  $B$  total bits and that  $S$  streams are used, then the probability of receiving exactly  $k$  of the  $B/S$  bits in each stream correctly is given by

$$p(k) = \begin{cases} \epsilon \cdot (1 - \epsilon)^k, & 0 \leq k < B/S \\ (1 - \epsilon)^k, & k = B/S \end{cases} \quad (5-2)$$

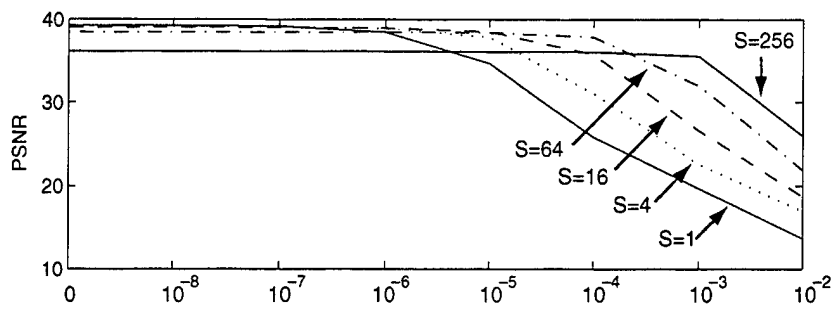
which is a valid probability mass function as one can easily verify by summing over all  $k$ . In Equation 5-2,  $(1 - \epsilon)^k$  is the probability that the first  $k$  bits are correct, while  $\epsilon$  is the probability that the  $(k+1)$ th bit is in error. Note that a separate term conditioned on  $B/S$  is necessary to take into account the possibility that all of the bits in the stream are correctly received. The mean value can now be calculated as

$$m_S = \sum_{k=0}^{B/S} k \cdot p(k) \quad (5-3)$$

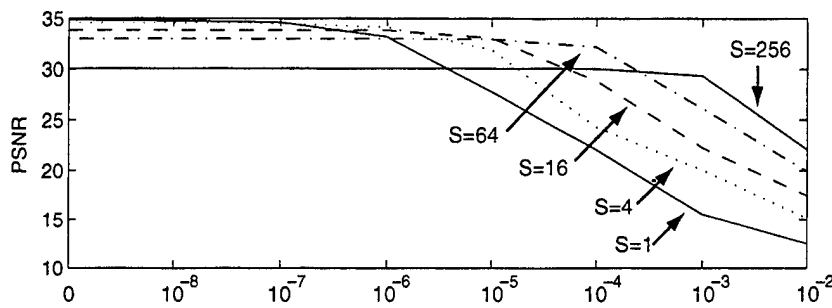
On the average, the total number of bits correctly received is  $S \cdot m$ . If  $B/S$  is large relative to  $1/\epsilon$ , then  $m_S \approx m_1$  for  $\forall S$ , and, therefore, approximately  $S$  times more bits are correctly decoded for the robust algorithm than for the conventional one (i.e.,  $S = 1$ ). Generally, the gain actually achieved is not this high, but it is nonetheless significant. In the "Results" below, we use Equation 5-3 to analyze the impact of transmission errors on the average quality of the reconstructed image for all possible values of  $S$ .

## RESULTS

For these comparisons, we introduce an error into all streams simultaneously according to Equation 5-3, and we allow each stream to self-terminate. Furthermore, we assume that the size of the image file is known by the decoder or, equivalently, that synchronization information is inserted after each image in a sequence. A 5-level transform using the 9/7 biorthogonal wavelet (Reference 5-9) is then applied to the image, and all possible robust partitionings are evaluated ( $S = 1$  is, again, the conventional EZW coder). From the objective results summarized in Figure 5-4, we note that considerable robustness is achieved at the expense of perfect channel rate-distortion performance. For the Lena image, the PSNR drops from 39.28 decibels ( $S = 1$ ) to 36.13 decibels ( $S = 256$ ), while for Barbara the equivalent drop is from 34.84 to 30.09 decibels. With an error rate of  $\epsilon = 10^{-2}$ , however, the average improvement of the REZW algorithm over a conventional EZW coder is more than 15 decibels for the same images. Examining Figure 5-5, we note that the objective performance of the ZP partitioning is generally superior to that achieved using offset zerotrees. This superiority is even more evident in the subjective quality, as shown by Figures 5-6 and 5-7. Most of the artifacts in Figure 5-7 are actually introduced by the arithmetic decoder overrun, and if this is eliminated (e.g., the position of the error is known), the ZP and OZ partitionings result in almost identical subjective quality at all error rates. Overrun distortion becomes less visually significant when more streams are used, as is illustrated by Figures 5-8 and 5-9, although it still reduces the PSNR more dramatically for the OZ partitioning. For a multi-channel transmission method such as orthogonal frequency division multiplexing, the offset ZP is perceptually superior to the ZP partitioning because the error rates in the different channels can be radically different. The advantages of the offset zerotree partitioning in this situation are clearly shown in Figures 5-10 and 5-11, where the truncation point in each stream has been separately calculated using Equation 5-3. Note that here we stop the coding process in each stream immediately after the first bit error occurs to prevent the decoder overrun from obscuring the comparison.



(a) Lena.



(b) Barbara.

FIGURE 5-4. PSNR (in dB) Versus the Probability of a Bit Error. Lena (a) and Barbara (b) coded at 1.0 bpp using ZP partitioning.

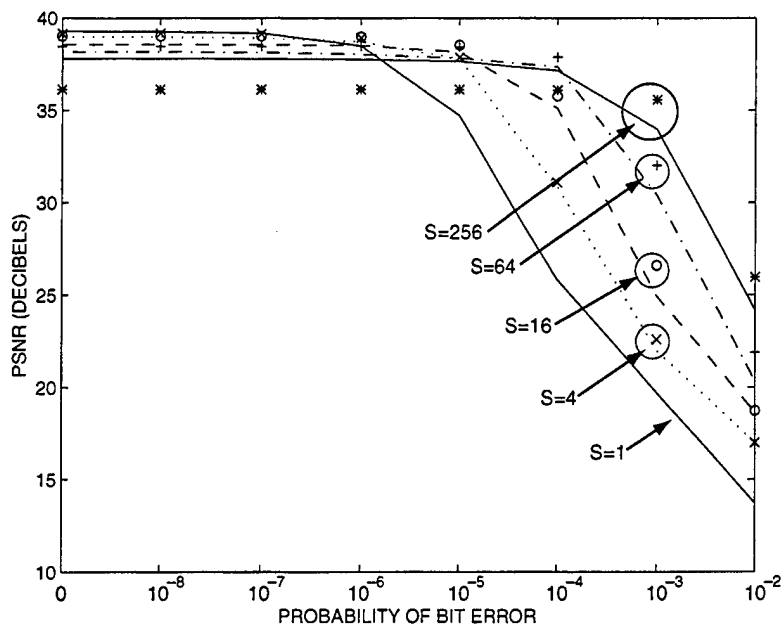


FIGURE 5-5. Comparison of Partitioning for Lena Image Coded at 1.0 bpp. Connected lines correspond to OZ partitioning, while symbols correspond to ZP partitioning. Note that ZP has slightly better rate-distortion performance in most cases, with the major exception being when  $S = 256$  and the error rate is low.



FIGURE 5-6. Lena Coded to 1.0 bpp Using ZP Partitioning With  $S = 4$  and Probability of Error  $\epsilon = 10^{-3}$ . PSNR = 22.57 dB.



FIGURE 5-7. Lena Coded to 1.0 bpp Using OZ Partitioning with  $S = 4$  and Probability of Error  $\epsilon = 10^{-3}$ . PSNR = 21.98 dB.



FIGURE 5-8. Lena Coded to 1.0 bpp Using ZP Partitioning With  $S = 256$  and Probability of Error  $\epsilon = 10^{-3}$ . PSNR = 35.57 dB.



FIGURE 5-9. Lena Coded to 1.0 bpp Using OZ Partitioning With  $S = 256$  and Probability of Error  $\epsilon = 10^{-3}$ . PSNR = 33.97 dB.



FIGURE 5-10. Lena Coded to 1.0 bpp Using ZP Partitioning With  $S = 4$  and Probability of Errors of  $10^{-2}$ ,  $10^{-3}$ ,  $10^{-4}$ , and  $10^{-5}$  in Each Stream. PSNR = 22.74 dB.



FIGURE 5-11. Lena Coded to 1.0 bpp Using OZ Partitioning With  $S = 4$  and Probability of Errors of  $10^{-2}$ ,  $10^{-3}$ ,  $10^{-4}$ , and  $10^{-5}$  in Each Stream. PSNR = 22.70 dB.

## IMPLEMENTATION AND COMPLEXITY

The robust compression algorithm illustrated by Figure 5-1 can be implemented efficiently either on a conventional sequential processor or on an array of MIMD parallel processors. A sequential implementation first performs a wavelet transform and then executes one of the vertical branches of Figure 5-1 until the appropriate number of bits (based on the interleaving structure) are produced. At that point, execution control is passed to the next branch, and the process continues in a round robin fashion until the bit allocation is exhausted. In the parallel implementation, a parallel wavelet decomposition is performed and each branch in the figure is executed independently by separate processors. In this case, either the outputs of the processors have to be synchronized to achieve proper interleaving or an additional output processor must be used to organize the data.

Neither implementation of the robust algorithm significantly affects the complexity of the complete system. Parallel wavelet transforms do have higher computational complexity (see Reference 5-10), but this increase is also incurred in a parallel implementation of the conventional algorithm (Reference 5-8). Furthermore, the search complexity of the REZW coder actually decreases slightly with  $S$  because the number of significant coefficients trends downward. The only quantifiable disadvantage of the robust algorithm is that it requires more temporary storage space because the coders operating in each stream must have their own thresholds and adaptive models. In reality, however, its most significant drawback is that it is simply more intricate than EZW, making it harder to efficiently program. Once programmed, it is just as fast as the conventional coder and requires only a small amount of additional memory.

## CONCLUSION

We have shown here how robustness to transmission errors can be added to an embedded image compression algorithm with little or no definable increase in its complexity. As the number of partitions,  $S$ , increases, the resilience of the coded image to transmission errors also increases, but the perfect channel rate-distortion performance of the codec decreases. If the bit error rate of the channel is greater than  $10^{-6}$ , however, the performance of our robust compression algorithm is generally better than that of the conventional embedded zerotree wavelet coder for some value of  $S$ .

## REFERENCES

- 5-1. T. M. Cover. "Broadcast Channels," *IEEE Trans. on Information Theory*, Vol. IT-18, No. 1 (January 1972), pp. 2-14.
- 5-2. J. M. Shapiro. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. on Signal Proc.*, Vol. 41, No. 12 (December 1993), pp. 3445-3462.
- 5-3. A. Said and W. A. Pearlman. "A New Fast and Efficient Image Codec Based on Set Partitioning Into Hierarchical Trees," *IEEE Trans. Circuits Syst. Video Tech.*, Vol. 6 (June 1996), pp. 243-250.
- 5-4. D. Taubman and A. Zakhor. "Multirate 3-D Subband Coding of Video," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 572-588.
- 5-5. C. D. Creusere. "Robust Image Coding Using the Embedded Zerotree Wavelet Algorithm," *Proc. Data Compression Conference* (March 1996), p. 432.
- 5-6. C. D. Creusere. "A Family of Image Compression Algorithms Which Are Robust to Transmission Errors," *Proc. SPIE*, Vol. 2825 (August 1996), pp. 890-900.
- 5-7. I. H. Witten, R. M. Neal, and J. G. Cleary. "Arithmetic Coding for Data Compression," *Commun. ACM*, Vol. 30 (June 1987), pp. 520-540.
- 5-8. C. D. Creusere. "Image Coding Using Parallel Implementations of the Embedded Zerotree Wavelet Algorithm," *Proc. SPIE*, Vol. 2668 (January 1996), pp. 82-92.
- 5-9. I. Daubechies. *Ten Lectures on Wavelets*, Philadelphia, PA, SIAM, 1992.
- 5-10. J. Lu. "Parallelizing Mallat Algorithm for 2-D Wavelet Transforms," *Information Proc. Letters*, Vol. 45 (1993), pp. 255-259.



## CHAPTER 6.

### INTERFRAME COMPRESSION

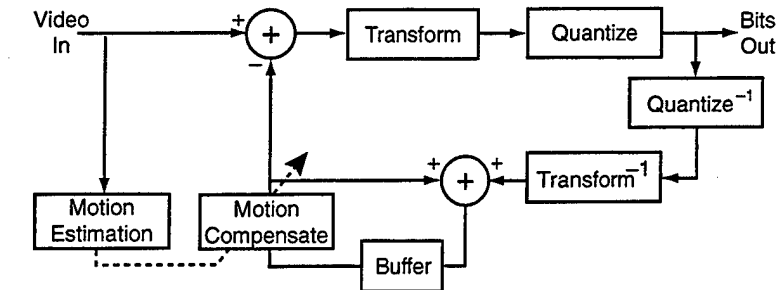
#### SUMMARY

In this chapter we consider the problem of compressing video for systems in which encoder complexity is a major constraint. While our area of particular concern is that of airborne remote sensing, the basic approach developed here is more widely applicable. Because of power constraints on a remote platform, video must be transmitted over narrow bandwidth communications channels, requiring that the encoder operate at very low bit rates. Consequently, a more complex encoder must be used that can exploit both spatial and temporal redundancy in the video sequence. To reduce this complexity, we introduce a new approach to motion compensation similar to the conventional hybrid differential pulse-code modulation (DPCM) transform method, but where the compensation is performed outside the feedback loop. Within this framework, many specific implementations are possible, and we study a few of them here. While the basic idea is conceptually similar to the pan compensation proposed by Taubman and Zakhor, our method continually tracks and updates the image in the feedback loop in the same way as the conventional hybrid coder. Using both residual energy and reconstruction error as metrics, we show that the new motion compensation scheme is actually superior to the conventional, one despite achieving an encoder complexity reduction of as much as 27%.

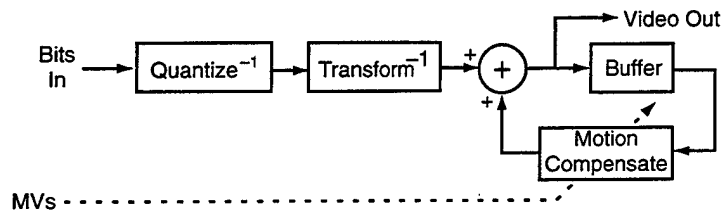
#### INTRODUCTION

In remote-sensing applications, severe constraints are placed on the design of the video encoder by weight, volume, and power limitations (Reference 6-1). To further complicate matters, power and antenna concerns force the communications channel through which the video must be transmitted to have a narrow bandwidth, requiring that the compression algorithm operate efficiently at low bit rates. In applications involving real-time remote control, video latency also becomes a significant systems constraint. Unfortunately, achieving good rate-distortion performance with such a restrictive channel generally requires a high complexity encoder and results in greater video latency, increasing the cost of the remote system and reducing its performance. For example, the most successful video compression technique to date, the hybrid DPCM-transform algorithm (Figure 6-1), has a far more complex encoder than decoder, exactly the opposite of what is desired in this application. To address this deficiency, we introduce a new method of motion compensation that eliminates the inverse transform operation required in Figure 6-1 and thus results in considerably lower encoder complexity than the

conventional approach (References 6-2 and 6-3). While we apply this new approach here only to sequences having primarily global platform motion, it might also be useful in other applications where complexity issues are critical, including mobile video teleconferencing.



(a) Encoder.



(b) Decoder.

FIGURE 6-1. Conventional Hybrid-Transform Video Compression System.

This chapter is organized as follows. We first discuss conventional solutions to the video compression problem, pointing out their disadvantages with respect to low complexity applications. Then our new method of motion compensated transform coding is introduced, along with a discussion of its trade-offs and practical implementation. We then detail the complete video coding algorithm and comparative results, followed by the conclusions.

## CONVENTIONAL MOTION COMPENSATION

### SPATIAL DOMAIN COMPENSATION

The most common form of hybrid-transform video compression performs the motion compensation and differencing operations in the spatial domain, as shown in Figure 6-1. This technique is used in all of the existing video compression standards, including the motion picture experts group (MPEG) I and II standards for broadcast (Reference 6-4 and 6-5) as well as the H.261 and H.263 standards for video teleconferencing (Reference 6-6 and 6-7). For these coders, the transform used in Figure 6-1 is an 8x8 blocked DCT. In terms of our application, the complexity of the encoder illustrated in Figure 6-1a is too high—at least double that of the corresponding decoder. The problem of encoder complexity is even worse for low bit rate applications because the relatively simple DCT is typically replaced with a more complex wavelet or subband decomposition. Doing this improves the rate-distortion performance of the video compression algorithm and eliminates many perceptually annoying artifacts (e.g., blocking), but it increases the encoder's complexity even more because the encoder contains both a forward and an inverse transform. Thus as better transforms are used to improve the performance of the spatial coder, the implementation cost of the encoder increases rapidly.

Assuming that  $F_k$  is the current frame in the video sequence, that an invertible transform is used, and that *no quantization* is applied, the reconstructed frame output from the decoder is

$$\tilde{F}_k = F_k - MC\{F_{k-1}\} + MC\{F_{k-1}\} \quad (6-1)$$

for the method of Figure 6-1, where MC indicates the motion compensation operator. Clearly, the frame reconstructed by the decoder is identical to that input to the encoder, regardless of what type of motion compensation is performed. This property offers flexibility that can be used to improve the rate-distortion performance of the system.

### TRANSFORM DOMAIN MOTION COMPENSATION

Many authors have proposed directly motion compensating the transform coefficients in many different ways and for a variety of reasons (References 6-8 through 6-14). In Reference 6-8, direct compensation of the coefficients was proposed so that spatial scalability could be introduced into the coded bit stream. The authors noted that the maximally decimated transforms typically used in coding applications are not spatially invariant and thus that the compensation operations must be altered when applied in the transform domain. Specifically, to compensate the coefficients in a given subband, one must, in general, use coefficients from all of the subbands in the image.

Despite achieving performance equivalent to that of spatial compensation, the method of Reference 6-8 is poorly suited to our needs because it also results in a significant increase in encoder complexity. Many of the other subband- or transform-based motion compensation schemes that have been proposed also increase the encoder's complexity (References 6-9 and 6-12).

Authors have also proposed the direct implementation of motion compensation in the subband domain (References 6-10 and 6-13). Figure 6-2 details the encoder of such a compression scheme, in which the image is first decomposed into different frequency bands using subband filters. Then motion compensation is performed in each band independently, using interpolation to compensate for decimation in the analysis filter bank. Where direct comparisons have been performed, however, motion compensation in the subband domain has often proven inferior to compensation applied in the spatial domain (Reference 6-14). To see why this degradation occurs, consider the simple 1-D 2-band filter bank shown in Figure 6-3. The operation of motion compensation can be implemented using an  $M$ -th band lowpass filter  $Q(z)$ , along with resampling operations as shown in Figure 6-4a, to compensate the subbands to  $2/M$  pixel accuracy with respect to the original image ( $d \cdot M/2$  is the shift relative to the original image). If  $M$  is odd, then Figure 6-4a can be redrawn using the noble identities as shown in Figure 6-4b (Reference 6-15). Examining this figure, we note that the anti-imaging filter  $Q(z)$  has now been replaced by  $Q(z^2)$ , and this allows high frequency replicas of the signal (created by upsampling) to pass through. Unfortunately, this analysis does not apply to the important case of single pixel shifts in which  $M$  equals 2. Nonetheless, one can easily confirm that the residual between an unshifted signal and one that has been shifted by  $d$  pixels and subsequently motion compensated using Figure 6-4 is never zero. Thus, despite its low complexity, the concept of direct compensation in the subband domain is seriously flawed.

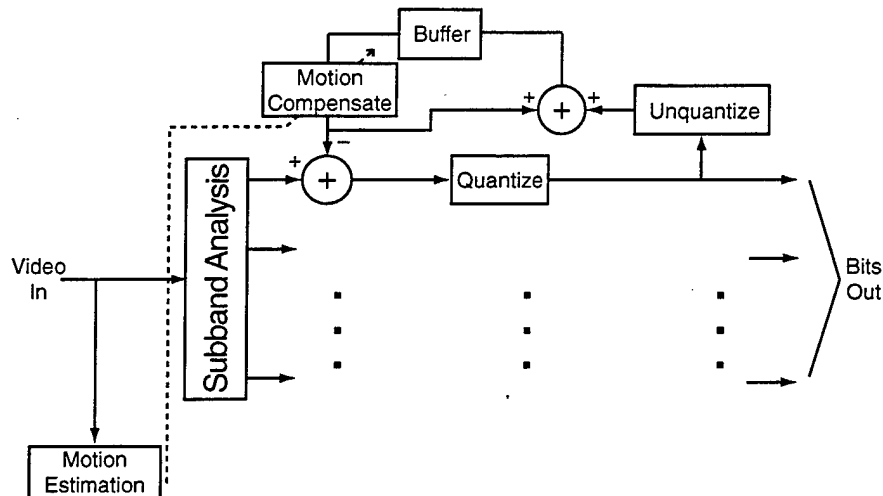


FIGURE 6-2. Video Encoder for In-Subband Motion Compensation.

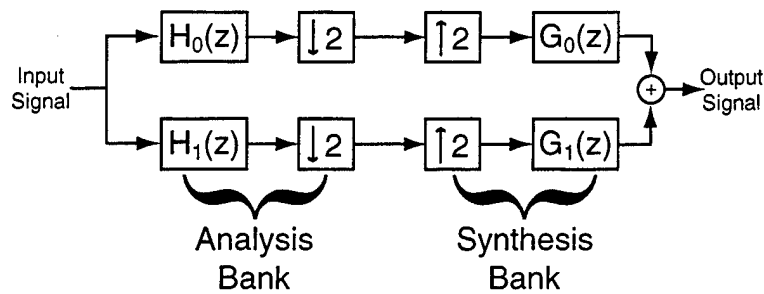
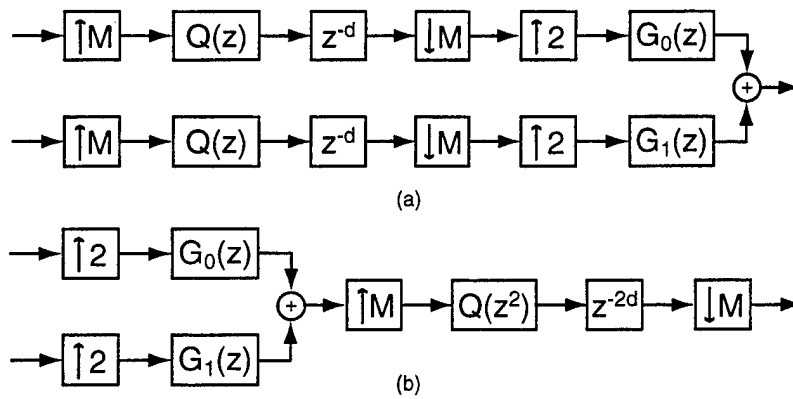


FIGURE 6-3. Two-Band Analysis/Synthesis Filter Banks.

FIGURE 6-4. Synthesis Filter Bank With In-Subband Motion Compensation ( $M/2$  Pixel Accuracy). (a) and (b) are equivalent.

## OUT-OF-LOOP MOTION COMPENSATION

### OVERVIEW

Figure 6-5 illustrates the structure of the proposed motion compensated video compression scheme (References 6-2 and 6-3). Note that the motion compensation is still performed in the spatial domain, as in the system of Figure 6-1, but that the differencing operation is now performed in the subband domain (as in Figure 6-2). Performing the differencing in the subband domain eliminates the need for the inverse transform in the feedback loop and, consequently, reduces the complexity of the video encoder by as much as 50% (depending on how the motion compensation and estimation are performed). Furthermore, if the bit stream produced by the quantizer is appropriately designed and partitioned, then the spatial resolution of the complete video coder is scalable (i.e., the decoder can retain in its feedback loop only those bits associated with the coarser wavelet scales (assuming that a wavelet-based algorithm is used, of course)

and still remain perfectly synchronized with the encoder in those scales). By performing the actual motion compensation in the spatial domain, we overcome the problems inherent in the shift-varying nature of maximally decimated filter banks and transforms, which were discussed above.

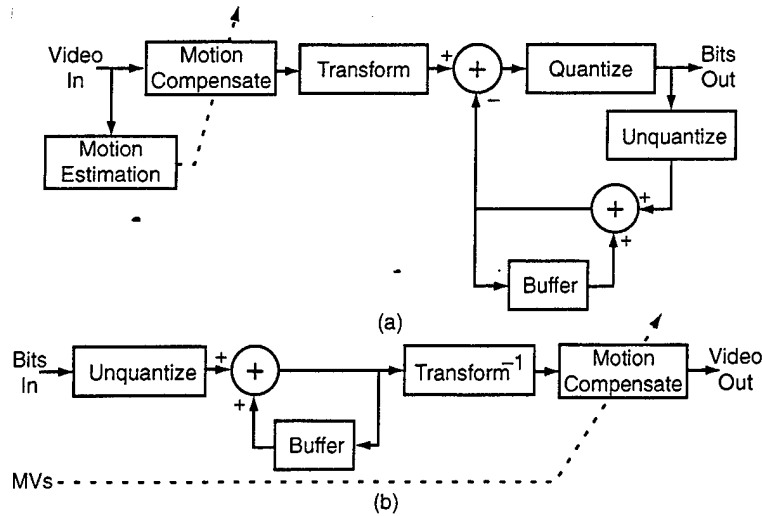


FIGURE 6-5. Hybrid Video Compression System Using Out-Of-Loop Compensation. (a) is the encoder and (b) is the decoder. Note that the motion vectors (MVs) are sent as side information to the decoder for correct motion compensation.

Unfortunately, there is a major drawback to the motion compensation structure of Figure 6-5. To see this, one can express the frame reconstructed by the decoder as

$$\tilde{F}_k = MC^{-1} \{ MC \{ F_k \} - F_{k-1} + F_{k-1} \} \quad (6-2)$$

assuming again, as in Equation 6-1, that no quantization is performed and that the transform is perfect reconstruction. Unlike the conventional motion compensation scheme characterized by Equation 6-1, our new method requires that the motion compensation operator be invertible. Previously, a motion compensated video compression algorithm with a structure similar to Figure 6-5 has been proposed by Taubman and Zakhor for use with panning (e.g., translational) motion (Reference 6-16). Their algorithm uses a 3-D subband decomposition and compensates for motion only over the support interval of the temporal filter bank, modifying the filters at the interval boundaries to achieve the invertibility required by Equation 6-2. In this work, we do not assume that the motion compensation is performed over a finite set of frames; instead, we require that the system continually track and update the motion estimate, regardless of the number of differentially predicted frames between each I-frame (e.g., a frame coded using only information contained within it).

If the original image is represented by an  $X \times Y$  matrix  $\mathbf{F}$  and the motion compensated image is represented by the matrix  $\hat{\mathbf{F}}$ , then global pan compensation can be viewed as a set of matrix multiplications, i.e.,

$$\hat{\mathbf{F}} = \mathbf{V} \cdot \mathbf{F} \cdot \mathbf{H} \quad (6-3)$$

where  $\mathbf{V}$  compensates for vertical motion and  $\mathbf{H}$  compensates for horizontal motion. Equation 6-2 requires that both  $\mathbf{V}$  and  $\mathbf{H}$  be invertible, and assuming this to be true, it becomes

$$\tilde{\mathbf{F}}_k = \mathbf{V}^{-1} \cdot \{ \mathbf{V} \cdot \mathbf{F}_k \cdot \mathbf{H} - \mathbf{F}_{k-1} + \mathbf{F}_{k-1} \} \cdot \mathbf{H}^{-1} \quad (6-4)$$

More generally, different  $\mathbf{V}$  and  $\mathbf{H}$  matrices can be used to shift each column and row of the image separately; we discuss this case in more detail below. Note that if the motion compensation is constrained to 1-pixel accuracy, then  $\mathbf{V}$  and  $\mathbf{H}$  are permutation matrices and their inverses are equal to their transposes. For the low complexity encoders of interest to us, this is a very important special case. Further, it is clear that rotational motion compensation can be performed within the framework of Equation 6-3 using the fast algorithm of Reference 6-17, and that the methods of Figures 6-1 and 6-5 should give equivalent results. The trade-offs associated with other forms of motion compensation satisfying Equation 6-2 are not so clearly defined. One possible course of action might be to search through all possible invertible compensation matrices  $\mathbf{V}$  and  $\mathbf{H}$  for each column and row of the image in order to find the set that minimizes the energy or entropy of the residual image (i.e., the image produced by subtracting the motion compensated input image from the previous image in Figure 6-5a). However this search process would be very cumbersome, and the resulting unstructured set of motion vectors would be very difficult to code. Consequently, the next section discusses practical ways in which motion compensation satisfying Equation 6-2 can be implemented and characterizes some of the associated trade offs.

## PRACTICAL IMPLEMENTATION

### Periodic Pan Compensation (PPC)

The largest single component of motion in many video sequences is panning motion. To compensate for such motion within the framework of Equation 6-2, we force pixels that have panned out of the image field to instead wrap around to the opposite edge (Reference 6-2). This is equivalent to assuming that the finite-extent image is instead an infinitely periodic image (i.e.,  $\mathbf{F}$  in Figure 6-6); thus, we call it periodic pan compensation (PPC). If  $F(x,y)$  is the  $X \times Y$  image to be compensated using the integer motion vector  $(\Delta m, \Delta n)$ , then PPC can be implemented as follows:

$$\text{PPC}\{F(x,y)\} = F((x - \Delta m) \bmod(X), (y - \Delta n) \bmod(Y)) \quad (6-5)$$

where “mod” is the standard modulo operation. Periodic pan compensation can also be described by Equation 6-3 with matrices

$$\mathbf{V}_{\Delta n, Y} = \begin{cases} \begin{bmatrix} \mathbf{0}_{\Delta n, Y-\Delta n} & \mathbf{I}_{\Delta n, \Delta n} \\ \mathbf{I}_{Y-\Delta n, Y-\Delta n} & \mathbf{0}_{Y-\Delta n, \Delta n} \end{bmatrix}, & \Delta n \geq 0 \\ \begin{bmatrix} \mathbf{0}_{Y+\Delta n, -\Delta n} & \mathbf{I}_{Y+\Delta n, Y+\Delta n} \\ \mathbf{I}_{-\Delta n, -\Delta n} & \mathbf{0}_{-\Delta n, Y+\Delta n} \end{bmatrix}, & \Delta n \leq 0 \end{cases} \quad (6-6)$$

and

$$\mathbf{H}_{\Delta m, X} = \begin{cases} \begin{bmatrix} \mathbf{0}_{X-\Delta m, \Delta m} & \mathbf{I}_{X-\Delta m, X-\Delta m} \\ \mathbf{I}_{\Delta m, \Delta m} & \mathbf{0}_{\Delta m, X-\Delta m} \end{bmatrix}, & \Delta m \geq 0 \\ \begin{bmatrix} \mathbf{0}_{-\Delta m, X+\Delta m} & \mathbf{I}_{-\Delta m, -\Delta m} \\ \mathbf{I}_{X+\Delta m, X+\Delta m} & \mathbf{0}_{X+\Delta m, -\Delta m} \end{bmatrix}, & \Delta m \leq 0 \end{cases} \quad (6-7)$$

where  $\mathbf{I}$  and  $\mathbf{0}$  are the identity and zero matrices, respectively, and the subscripts indicate the matrix dimensions. Note that the periodic extension shown in Figure 6-6 is the same as that used to implement circular convolution in Reference 6-18. Thus, if the subband or wavelet filters are circularly convolved with the image, then using PPC will not adversely affect the transform coding gain. To compensate for translation, a square window defining the image fed into the transform in Figure 6-5a is moved around the infinite extent image of Figure 6-6 as appropriate for the motion. Clearly, periodic pan compensation as shown in Figure 6-6 will perform less efficiently in the part of the window that extends outside the center repetition of  $F$  (e.g., the original finite-extent image) because there is likely to be less correlation between the wavelet coefficients generated by these pixels and the equivalent coefficients maintained in the feedback loop. We note, however, that this loss of efficiency does not accumulate with succeeding frames of the video sequence. After quantization and coding, an approximation of the region outlined by the square box in Figure 6-6 becomes the reference image. If the next input frame is shifted as indicated by the gray box, then *only* the region shown in gray will have a higher residual energy (and, likely, reduced coding performance) because it represents the area in which the two frames do not overlap.



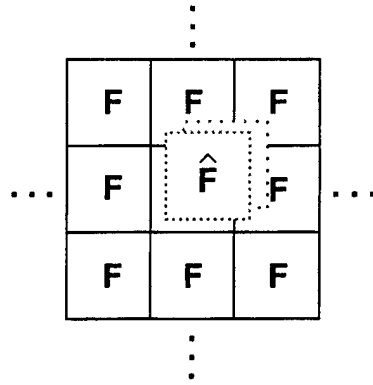


FIGURE 6-6. Diagram of Periodic Pan Compensation, Where  $F$  is the Periodically Repeated Original Image and  $\hat{F}$  is the Compensated Image Input to the Transform.

By making a few reasonable assumptions about the statistics of the video sequence, we can compare pan compensation implemented with the conventional spatial differencing of Figure 6-1 to that implemented using the out-of-loop approach shown in Figure 6-5. Because pan compensation is implemented separably in the horizontal and vertical directions, it suffices here to consider only a 1-D system. In addition, because the adverse effects of PPC are not cumulative, we need only consider the compensation of a single frame. Assume that the input  $x$  is a random vector in  $\mathbf{R}^N$  with

$$\text{mean} = E\{x\} = 0 \quad (6-8)$$

and

$$\text{cov} = E\{x \cdot x^t\} = \begin{bmatrix} \sigma^2 & \alpha \cdot \sigma^2 & \alpha^2 \cdot \sigma^2 & \dots \\ \alpha \cdot \sigma^2 & \sigma^2 & \alpha \cdot \sigma^2 & \alpha^2 \cdot \sigma^2 \\ \alpha^2 \cdot \sigma^2 & \alpha \cdot \sigma^2 & \sigma^2 & \alpha \cdot \sigma^2 \\ \dots & \alpha^2 \cdot \sigma^2 & \alpha \cdot \sigma^2 & \sigma^2 \end{bmatrix} \quad (6-9)$$

or to put it another way,

$$\text{cov}_{j,k} = E\{x(j) \cdot x(k)\} = \alpha^{|j-k|} \cdot \sigma^2 \quad (6-10)$$

where  $E\{\bullet\}$  is the expectation operator and  $0 \leq \alpha < 1$ . We further assume that the previous input vector  $\hat{x}$  also satisfies Equations 6-8 and 6-9 with the additional condition on the cross correlation that

$$E\{x(j) \cdot \hat{x}(k)\} = \alpha^{|j-k+\Delta n|} \cdot \sigma^2 \quad (6-11)$$

assuming that the input vector is shifted by  $\Delta n$  with respect to  $\hat{\mathbf{x}}$ . If the new input truly is just a shifted version of the previous input, then the assumption that  $\hat{\mathbf{x}}$  also satisfies Equations 6-8 through 6-11 is accurate. Equation 6-9 states that the statistical correlation between elements in the signal decreases as the distance between them increases. This common statistical image model is known to hold well over limited regions of an image, but we apply it broadly here in order to characterize the worst case performance of PPC relative to conventional pan compensation.

The pan compensation for Figure 6-1 can also be formulated as a matrix multiplication similar to Equation 6-3 with

$$\mathbf{V}'_{\Delta n} = \begin{cases} \begin{bmatrix} \mathbf{1}_{\Delta n,1} & \mathbf{0}_{\Delta n,N-1} \\ \mathbf{I}_{N-\Delta n,N-\Delta n} & \mathbf{0}_{N-\Delta n,\Delta n} \end{bmatrix}, & \Delta n \geq 0 \\ \begin{bmatrix} \mathbf{0}_{N+\Delta n,-\Delta n} & \mathbf{I}_{N+\Delta n,N+\Delta n} \\ \mathbf{0}_{-\Delta n,N-1} & \mathbf{1}_{-\Delta n,1} \end{bmatrix}, & \Delta n \leq 0 \end{cases} \quad (6-12)$$

where  $\mathbf{1}$  is a matrix of all ones. Note that we use here the freedom provided by Equation 6-1 to extend the compensated vector (or, in general, image) by repeating elements in regions uncovered by the motion: given our statistical model, this is the best course of action. The residual vector is given by

$$\boldsymbol{\varepsilon} = \mathbf{x} - \hat{\mathbf{x}} \quad (6-13)$$

and the corresponding expected squared error by

$$E\{f\} = E\{\boldsymbol{\varepsilon}^t \cdot \boldsymbol{\varepsilon}\} = E\{(\mathbf{x} - \hat{\mathbf{x}})^t \cdot (\mathbf{x} - \hat{\mathbf{x}})\} \quad (6-14)$$

where  $(\bullet)^t$  is the transpose operation and  $f$  is simply the sum of the squared errors. Simplifying Equation 6-14 using the identity  $\mathbf{x}^t \cdot \mathbf{x} = \text{trace}(\mathbf{x} \cdot \mathbf{x}^t)$  and the linearity of the trace and expectation operators, we find that

$$\begin{aligned} E\{f_c\} &= 2N \cdot \sigma^2 - 2 \cdot \text{trace}(E\{\mathbf{V}'_{\Delta n} \cdot \hat{\mathbf{x}} \cdot \mathbf{x}^t\}) \\ &= 2N \cdot \sigma^2 - 2 \cdot \text{trace}(\mathbf{V}'_{\Delta n} \cdot E\{\hat{\mathbf{x}} \cdot \mathbf{x}^t\}) \end{aligned}$$

where  $f_c$  is the error in the case of conventional pan compensation. Finally, applying Equation 6-11 and simplifying,

$$E\{f_c(\alpha, \Delta n)\} = 2\sigma^2 \cdot \left( |\Delta n| - \frac{\alpha}{1-\alpha} (1 - \alpha^{|\Delta n|}) \right) \quad (6-15)$$

which of course equals zero if  $|\Delta n| = 0$ .

Starting with Equation 6-14 and using Equation 6-6 in place of (6.12), we can write a similar expression for the case in which periodic pan compensation is used, i.e.,

$$E\{f_{\text{PPC}}\} = 2N \cdot \sigma^2 - 2 \cdot \text{trace}(\mathbf{V}_{\Delta n, N} \cdot E\{\hat{\mathbf{x}} \cdot \mathbf{x}^t\}) \quad (6-16)$$

where  $\mathbf{V}$  now performs a full circular shift of the rows of the correlation matrix. Applying Equation 6-11, circularly shifting, and evaluating the trace, we find that

$$E\{f_{\text{PPC}}(\alpha, \Delta n)\} = 2\sigma^2 |\Delta n| \cdot (1 - \alpha^N) \quad (6-17)$$

The difference between Equations 6-17 and 6-15 represents the loss of efficiency incurred by using PPC over conventional compensation and is given approximately by

$$\text{diff} = \frac{2\sigma^2 \cdot \alpha}{1 - \alpha} \{1 - \alpha^{|\Delta n|}\} \quad (6-18)$$

for large  $N$  and reasonable values of the regional correlation coefficient  $\alpha$ . To characterize the performance degradation graphically, we form the ratio

$$\text{ratio} = \frac{N \cdot \sigma^2 - \text{diff}}{N \cdot \sigma^2} = 1 - \frac{2\alpha}{N \cdot (1 - \alpha)} (1 - \alpha^{|\Delta n|}) \quad (6-19)$$

and plot this in Figure 6-7 for  $N = 512$  (the length of one line in a typical image) and a few  $\alpha$ s. This ratio basically characterizes the increase in residual energy relative to the energy of the original input signal, and we use it here to eliminate the dependence on  $\sigma^2$ . From the figure, we note that the penalty paid for using PPC (i.e., the reduction from 1.0) is not truly severe even for large values of  $\Delta n$  and  $\alpha$ , leading us to expect that the method will perform well in practice. The performance of PPC relative to conventional pan compensation improves as the regional correlation in the statistical image model decreases because the conventional algorithm's flexibility in filling the part of the frame uncovered by motion becomes less significant.

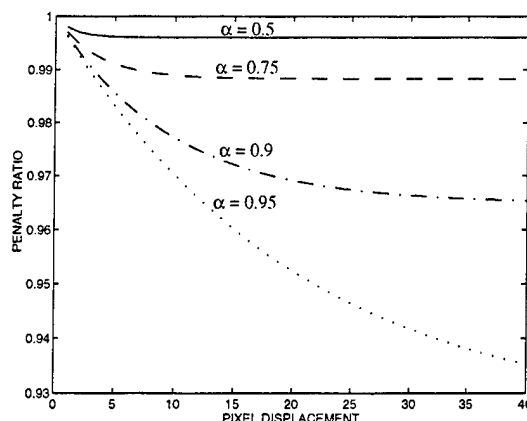


FIGURE 6-7. Difference in the Expected Error Between Conventional and Periodic Pan Compensation for a Variety of  $\alpha$ .

### Periodic Generalized Pan Compensation (P-GPC)

The concept of periodic pan compensation can be easily generalized to allow for more complex motion while retaining the same basic framework (Reference 6-3). Specifically, each individual row and column of the image can be separately compensated, as shown in Figure 6-8. This flexibility allows one to compensate for much more complex global motions including rotation (using the method proposed in Reference 6-17). In the context of P-GPC, Equation 6-5 can be rewritten as

$$P-GPC\{F(x,y)\} = F((x - \Delta m_x) \bmod(X), (y - \Delta n_y) \bmod(Y)) \quad (6-20)$$

where  $\Delta m_x$  and  $\Delta n_y$  are the individual motion vectors corresponding to each row and column, respectively. Note that the analysis of PPC discussed previously is also applicable here because it only considers a single line of the image (an average over all lines can be computed for P-GPC to estimate the performance decrease for the entire image).

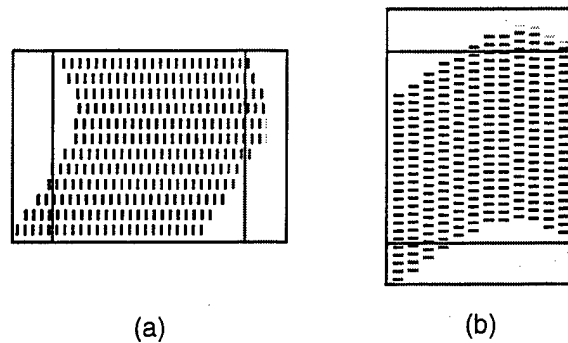


FIGURE 6-8. Generalized Periodic Pan Compensation for Horizontal (a) and Vertical (b) Motion. The dashed lines are the motion compensated lines and the gray areas are where the lines wrap around.

### Region-Based Motion Compensation

The concepts discussed above can also be applied over specified regions within the image. Doing this compensates for motion locally, which can be advantageous when the image contains multiple, independently moving objects. But the pixel wraparound required to satisfy Equation 6-2 can greatly increase the residual error if the regions are small relative to the amount of motion. Because the amount of pixel wraparound accumulates with successive frames, the sizes of the motion compensation regions and the average amount of motion in them act to limit the number of consecutive frames that can be coded before the error residual begins to increase dramatically. In practice, if this form of region-based motion compensation is applied to an unknown image sequence, the encoder must closely monitor the error residual and insert an I-frame (or a portion of one over a region) when it grows too large. Finally, one should note that at very low bit rates the block-circular shifting at the decoder's output is likely to introduce false edges into the resulting video. These artifacts will be present even if a transform having overlapped basis functions (e.g., a wavelet) is used.

### Compensating for Subpixel Motion

So far we have only considered systems capable of compensating for integer amounts of video motion. While integer-accurate compensation is well suited to applications where complexity is critical, the energy of the residual image can often be further reduced by compensating for subpixel motion. To minimize computational complexity, we initially considered compensation based on a simple linear interpolation filter. Unfortunately, such filters have zeros at  $z = -1$  in the complex plane, making stable inversion (a requirement in our application) impossible. Therefore, we have adopted the approximately allpass interpolator developed in Reference 6-19. Equation 6-20 can be modified to include subpixel interpolation as follows:

$$S - GPC\{F(x, y)\} = q_{\beta(x), \beta(y)} \left[ F\left((x - \Delta m_x) \bmod(X), (y - \Delta n_y) \bmod(Y)\right) \right] \quad (6-21)$$

where  $q[k]$  represents the infinite impulse response interpolation filter and  $\beta(\cdot)$  is the sub-integer shift (i.e.,  $\beta(\cdot) \in [-1/2, 1/2]$ ) for the specified row or column. In our application of the allpass interpolator, we have found that a window size of 10 along with three iterations of Equation 6-12 in Reference 6-19 provide adequate performance. One should note, however, that even with these simplifications the computational complexity of this approach is still quite high.

## VIDEO CODING ALGORITHM

### MOTION ESTIMATION

To minimize complexity in the video encoder, one would prefer to estimate motion vectors using ancillary information such as sensor gimbal movements and platform navigation information. Unfortunately, such information does not exist for the video sequences available to us. Therefore the video coding algorithm presented here (and used to generate the results below) also performs motion estimation, but we note that this added computational cost may not be germane to the final application.

All of our motion estimates are made by comparing 128x128 regions of the current video frame with the same regions in the previous frame. It is necessary to actually store portions of the previous frame because the out-of-loop compensation structure of Figure 6-5 never reconstructs the actual image produced by the decoder (i.e., it only reconstructs the decoder's approximation of the transform coefficients). Integer accuracy motion estimates for each region are then formed using FFT-based correlation between the frames. We selected this method of matching because it tended to provide better estimates than spatial searches for our video sequences. To compute the motion estimates for pan compensation, we use a single region centered in each video frame, while for the two other methods we use five different regions. Figure 6-9 shows the correlation regions used for the GPC method as well as those used with a region-based motion compensation method of the type discussed previously. For GPC, a different horizontal (vertical) motion vector is calculated for each row (column) by interpolating the five motion vectors generated by correlation in Figure 6-9a. For example, the horizontal motion vector for line  $\delta$  in the figure is determined by first averaging the horizontal motions of regions 0, 1, and 3 (the magnitude of which is indicated by the dashed arrow) and linearly interpolating this average with the horizontal component of the motion vector in region 4. This process is extrapolated outward past the center of region 4 (along the dotted line) to estimate the motion vector for line  $\gamma$ . All other horizontal and vertical motion vectors are computed in the same way using vector components from the appropriate regions. In Figure 6-9a, our implementation of GPC assumes a smooth, piecewise-linear motion

model. We have found that smoothness is especially important in achieving good spatial compression if a transform having overlapped basis functions is used, a fact also noted in Reference 6-9.

Motion estimates for the region-based compensation approach are calculated using the dashed squares shown in Figure 6-9b. In this case, the motion vector for region 0 ( $\mathbf{mv0}$ ) is used to estimate the global panning motion, while those of regions 1 through 4 ( $\mathbf{mv1}, \dots, \mathbf{mv4}$ ) are used to estimate the motion in the associated quadrant of the image. To compensate the current video frame, this encoder first uses  $\mathbf{mv0}$  to perform periodic pan compensate on the entire frame and then uses the differences ( $\mathbf{mv1}-\mathbf{mv0}$ ,  $\mathbf{mv2}-\mathbf{mv0}$ ,  $\mathbf{mv3}-\mathbf{mv0}$ , and  $\mathbf{mv4}-\mathbf{mv0}$ ) to perform PPC in each quadrant separately. For comparison, we also use the same motion estimates to compensate a conventional hybrid DPCM-transform coder but with pixel replication to fill in the uncovered regions.

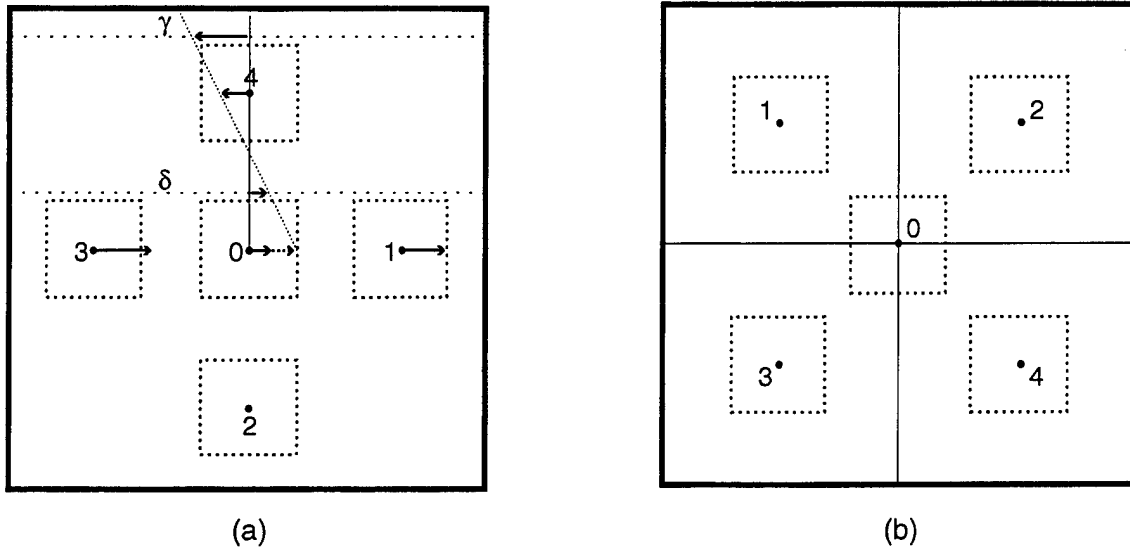


FIGURE 6-9. (a) Shows Motion Estimation Method for Generalized Pan Compensation; (b) Shows Motion Estimation for Region-Based Approach. Small dashed squares are the regions used in each frame for correlations.

## TRANSFORM-BASED RESIDUAL ENCODER

To evaluate the proposed motion compensation algorithm, we use a 5-level, 2-D wavelet transform that has been implemented separably with 1-D 9/7 biorthogonal wavelets (9 and 7 tap analysis low and high pass filters, respectively) (Reference 6-20). Symmetric extension is applied to the wavelet filters at image borders except as noted in Figure 6-10, where circular convolution is also used (Reference 6-20). For rate-distortion comparisons, we use the EZW algorithm to code and quantize both the I- (intra) and P- (predicted) frames in the sequence (Reference 6-21). In addition to achieving good rate-distortion performance, this algorithm generates an embedded bit stream (i.e., information is transmitted in order of importance) that, in a complete video coder,

simplifies the rate-buffer control. While functional, the video coding algorithm specified here has been primarily designed for the purpose of comparing the various motion compensation schemes previously discussed. For the results presented below, we use fixed bit allocations of 0.2 bpp for the first image in the sequence (the I-frame) and 0.1 bpp for all of the subsequent difference frames (P-frames). A complete low bit-rate video coder might also make use of bi-directionally motion compensated predicted and/or interpolated frames (B-frames), and it could possibly also vary the bit allocation to different frames in the sequence in an effort to maintain uniform quality in the reconstructed video sequence. These enhancements would greatly increase the performance of the basic video coding algorithm described here but only at the expense of increased encoder complexity.

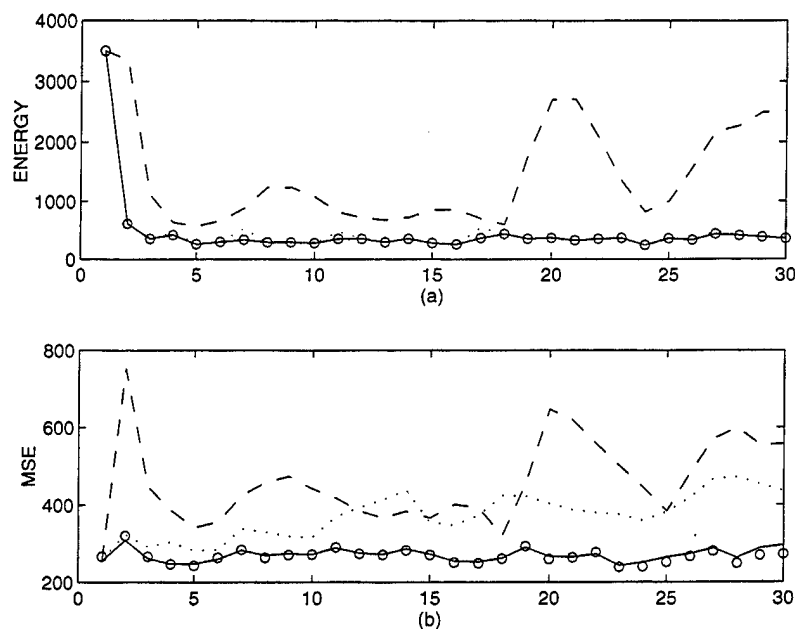


FIGURE 6-10. (a) Energy of Difference Image (No Quantization) Versus Frame Number; (b) Mean Squared Error Of Reconstruction With Quantization Versus Frame Number. Dashed is non-motion compensated, solid is PPC with symmetric extension, circles are PPC with circular convolution, and dotted is conventional pan compensation.

## RESULTS

The video to be used for these comparisons is an aerial sequence of a power plant consisting of 30 512x512 frames collected with an imaging infrared camera. In our comparisons, we process only one field of each frame (i.e., every other line of an image) to eliminate any degradation in coder performance due to the camera's interlaced scan. The first and last frames of the sequence are shown in Figures 6-11 and 6-12,



respectively, and the results of the comparisons are plotted in Figures 6-10, 6-13, and 6-14 for the different motion compensation methods proposed earlier. In these plots, "ENERGY" refers to the sum of the squared pixel values divided by the total number of pixels, and "MSE" refers to the mean squared error between the original image and its reconstruction (using the coding algorithm specified above). Studying Figure 6-10a, we note that PPC in the context of Figure 6-5 and conventional pan compensation in the context of Figure 6-1 both provide considerable reductions in residual energy over the uncompensated difference image (dashed line). In fact, the residual energy reductions achieved with any form of motion compensation are very similar. Comparisons of the MSEs, however, show that PPC combined with the out-of-loop structure performs far better than conventional pan compensation as the frame number increases (the average MSE is 270 versus 371). The same phenomena also occur with generalized pan compensation (Figure 6-13) and block-based compensation (Figure 6-14). Why? We find that the energy of the transformed residual image is concentrated into fewer coefficients when an out-of-loop video encoder is used in place of a conventional one. The poor energy concentration of the conventional hybrid DPCM-transform coder is a direct result of spatial energy spreading in the transform: i.e., the residual energy is contained in a few large pixels after the subtraction of Figure 6-1, but the transform spreads it out over a number of wavelet scales (subbands) and a number of coefficients in a given scale. Thus, the zerotree spatial encoder must transmit more significant coefficients, making its entropy coding (the zerotree root symbol plus the arithmetic encoder) less effective. Also plotted in Figure 6-10 is the case in which circular convolution is applied at the image borders in place of symmetric extension; we note that circular convolution is slightly better (as predicted previously) but not by very much: its average MSE is 267 versus 270 for symmetric extension.

Figure 6-13 compares the out-of-loop and conventional motion compensation methods using generalized pan compensation (periodically implemented in the first case) as presented earlier, with the motion estimation methodology described above. These results are very similar to those of Figure 6-10, with the out-of-loop compensation scheme offering the best rate-distortion performance (average MSE of 273 versus 337). In Figures 6-15 and 6-16 one also notes that by the end of the sequence, the reconstructed video has degraded much more severely for the conventional video coder than it has for the out-of-loop coder. Subpixel P-GPC (using the scheme discussed earlier) is also shown in Figure 6-13 (circles). Comparing this to integer P-GPC we find that the average MSE decreases by only three, but this is not surprising because the input image sequence appears to have been spatially sampled at a relatively high rate compared to its actual resolution. In addition, the MSE is increased by between 5 and 15 by the inverse subpixel interpolation filter because it is imperfect near the image borders.

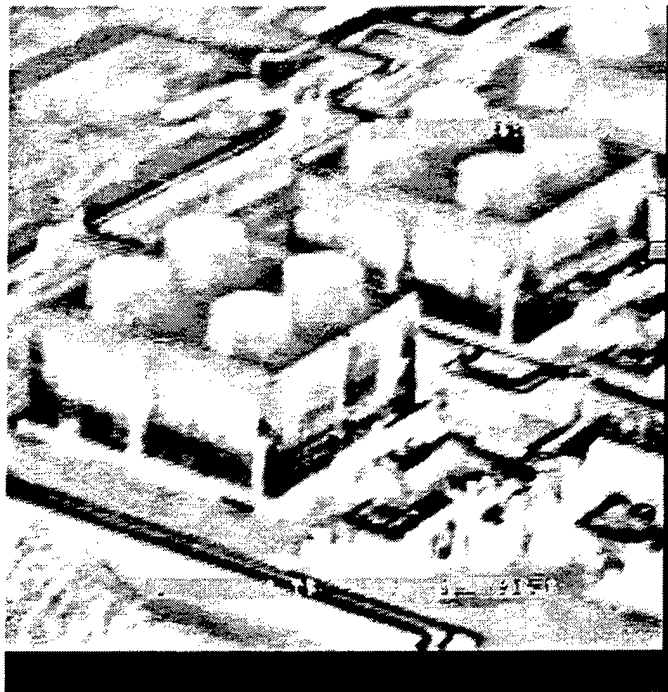


FIGURE 6-11. First Frame of Image Sequence.

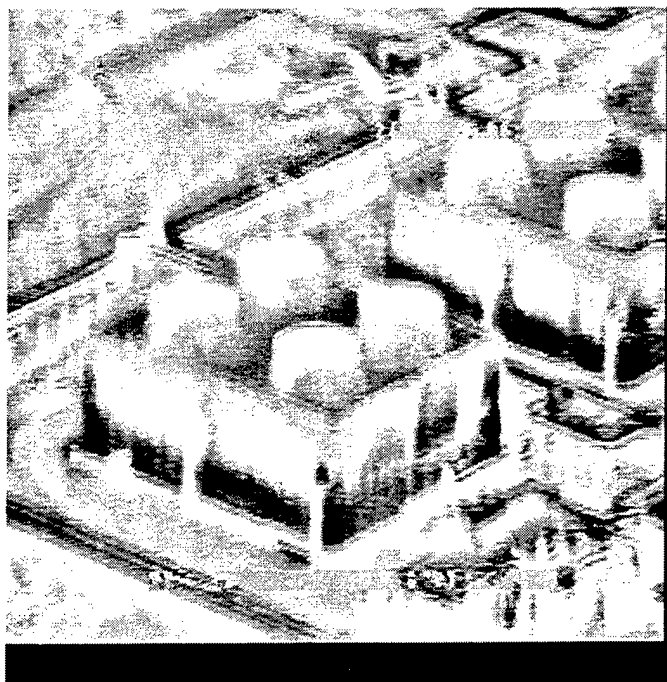


FIGURE 6-12. Last (30th) Frame of Image Sequence.

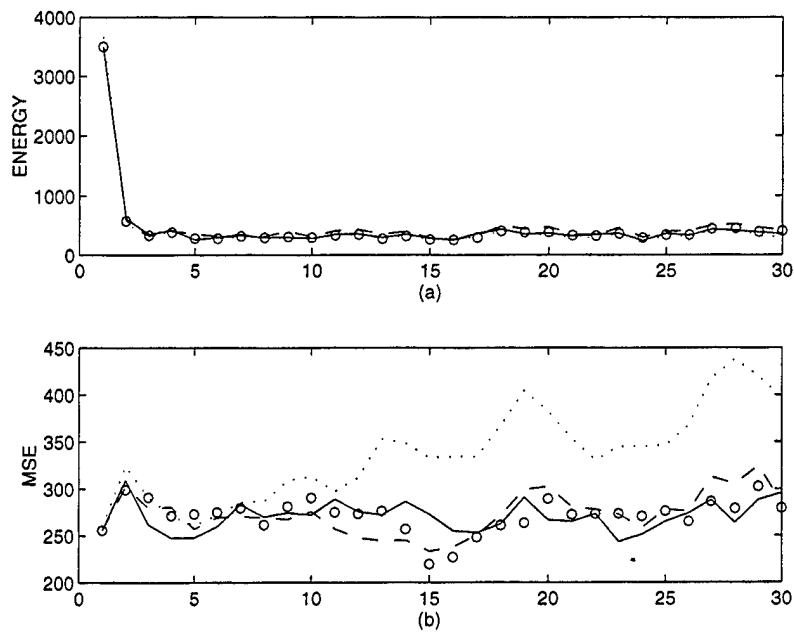


FIGURE 6-13. (a) Difference Energy and (b) MSE Versus Frame Number. Solid is repeated from Figure 6-10, for reference, dashed is generalized periodic pan compensation, circles are subpixel P-GPC, and dotted is non-periodic GPC implemented within a conventional hybrid DPCM-transform framework (i.e., Figure 6-1).

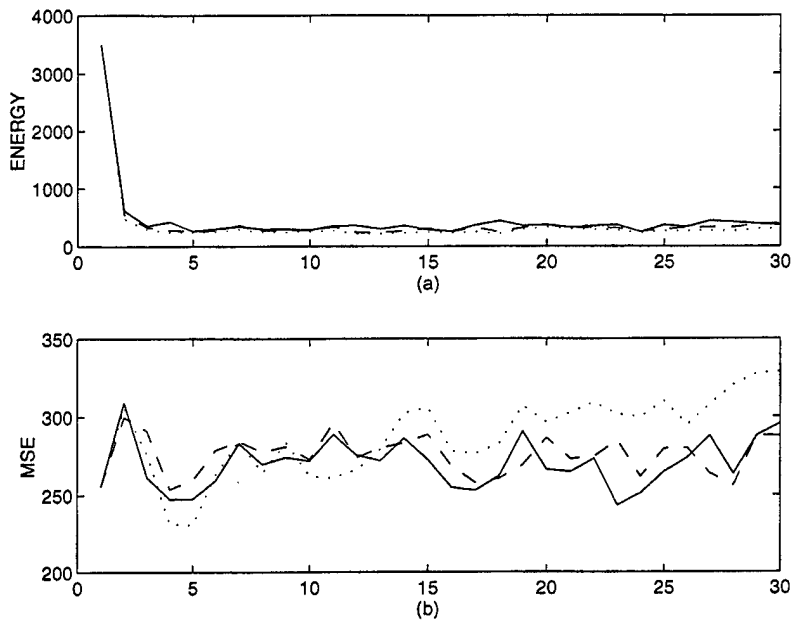


FIGURE 6-14. (a) Difference Energy and (b) MSE Versus Frame Number. Solid line is repeated from Figure 6-12, dashed is region-based motion compensation using PPC and the "out-of-loop" framework (i.e., Figure 6-5), dotted is region based using the conventional framework.

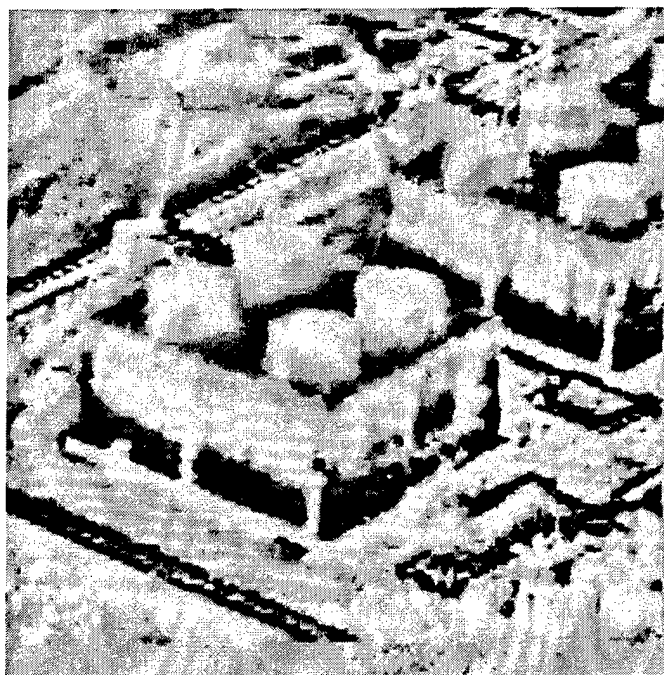


FIGURE 6-15. Frame 30 Using Out-Of-Loop Compensation With P-GPC.

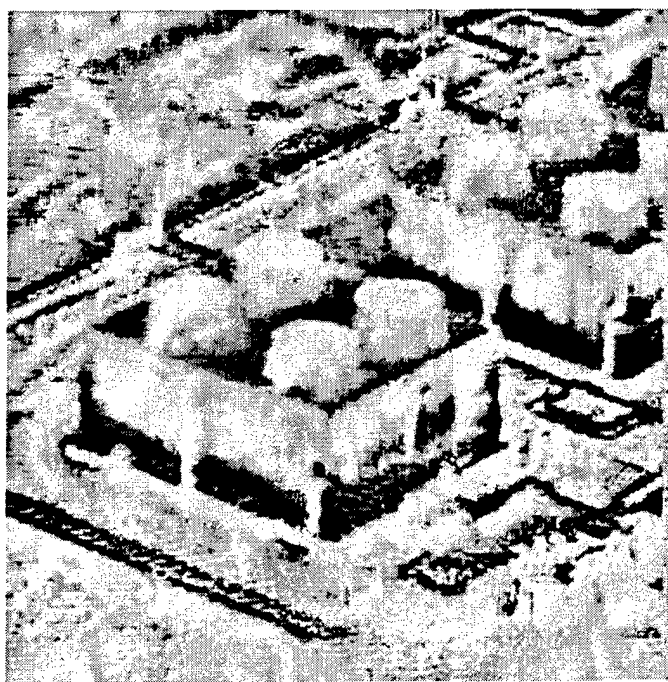


FIGURE 6-16. Frame 30 Using Conventional Approach With GPC.

Comparisons for the block-based motion compensation approach (discussed above) are illustrated objectively by Figure 6-14 and subjectively by Figures 6-17 and 6-18. Again, we note that the energy residuals for the two methods are almost identical while the MSE of the out-of-loop compensation approach is better (276 versus 287). For this sequence, nothing is gained by using a block-based approach over simple pan compensation, indicating that most of the motion in the sequence is translational. Note that at the bit rates used here, some artifacts do occur in the reconstructed image sequence using both the out-of-loop and conventional approaches. Unfortunately, these artifacts tend to be most noticeable for the out-of-loop motion compensation. This happens because the discontinuities introduced by block-based motion compensation tend to be smoothed over by the coding process in the conventional approach, while for the out-of-loop method they are introduced directly into the reconstructed video sequence.

The results presented thus far have shown that out-of-loop motion compensation achieves rate-distortion performance superior to that of conventional hybrid DPCM-transform coding when used in a wavelet-based, fixed rate system. While this is excellent, we are still very interested in the encoding complexity: reducing this complexity is the key to using interframe coding techniques in a variety of remote video applications. With our aerial video sequence, we found above that simple pan compensation extracted most of the interframe redundancy. Thus we compare here the complexities of the out-of-loop and the conventional approaches only within the context of pan compensation. To perform these comparisons, we have timed the execution of each algorithm on a Sun Sparc 10 computer; and to analyze the effects of algorithm optimization, we have run each case twice: once when compiled with no runtime optimization and next when compiled with the maximum amount of optimization. Without optimization, the new encoder takes 651 seconds to process the 30 frames in the sequence, versus 899 seconds for the conventional encoder, resulting in a 27.6% speed increase. Compiler optimization reduces the encoding times to 388 seconds and 481 seconds for the new and conventional algorithms, respectively—a 19.3% speed increase. Note that this speed increase is achieved despite our use of two fairly large discrete Fourier transforms (256x256 with frequency domain interpolation) to estimate image motion for each frame. If a more efficient estimation technique were used, the speed increases would be much larger. Decoder times for the new and conventional algorithms are 401 seconds (212 seconds with optimization) and 404 seconds (217 with optimization), respectively. Thus, the decoding complexities of the two algorithms are essentially identical, as expected.

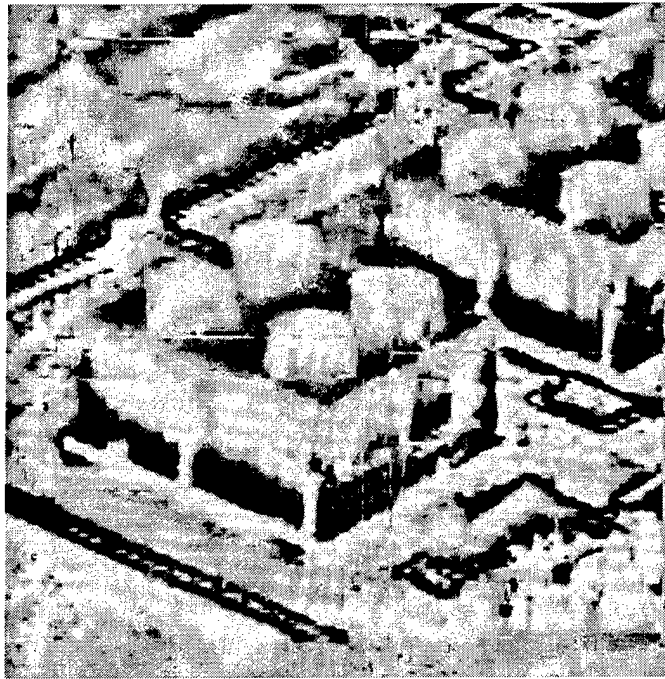


FIGURE 6-17. Frame 30 Using Out-Of-Loop Compensation With Blocks.

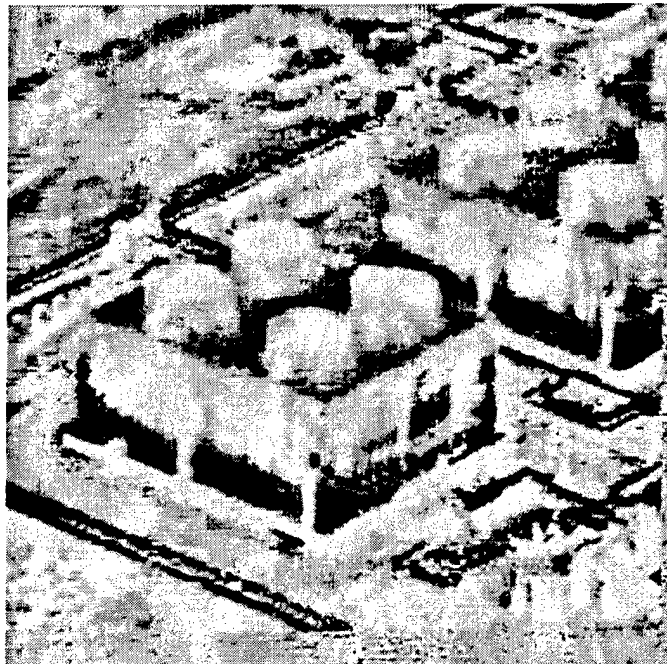


FIGURE 6-18. Frame 30 Using Conventional Approach With Blocks.

## MAINTAINING ROBUSTNESS TO TRANSMISSION ERRORS

In Chapter 5 we introduced an embedded compression algorithm that is inherently robust to transmission errors. Unfortunately, this algorithm only extracts the redundancy within single video frames (i.e., it does not exploit in any way the large correlation between adjacent temporal frames in a video sequence). By exploiting such temporal correlation, one can greatly improve compression performance but at the expense of transmission error robustness. For example, within the compression frameworks of Figures 6-1 and 6-5, a single transmission error can cause catastrophic error propagation because it de-synchronizes the feedback loops of the encoder and decoder. Conventionally, the only way to recover from this situation is to send I-frame (intraframe coded) updates at regular intervals. This has the effect of reducing the compression system's rate-distortion performance. We propose here an alternative to I-frame updates that is highly compatible with the REZW still-frame coder discussed in Chapter 5. Figure 6-19 illustrates the proposed robust real-time video compression algorithm where the "embedded coder" and "embedded decoder" blocks incorporate our basic REZW partitioning approach. To speed up the encoding process, the decoder's wavelet coefficient approximations are never directly calculated by the encoder; rather, the error residual after encoding is simply added to or subtracted from (depending on the sign of the predicted coefficient) the actual predicted coefficient's value (i.e., after subtraction) to calculate these approximations. The key to achieving robust transmission is the use of "leaky" prediction. If the weighting factor  $W$  is set equal to 1, then we have conventional predictive coding. If, on the other hand, it is selected to be less than 1, then we allow "leakage" of the original image into the residual. This leakage allows the decoder to forget coefficient errors that have been introduced into its frame buffer (by faulty transmission or other means). Thus, if an error occurs in the transmitted bit stream, REZW encoding ensures that the error does not propagate spatially, while leaky prediction ensures that it does not propagate in time. Note that the proposed robust video compression algorithm is also fully compatible with the out-of-loop motion compensation illustrated by Figure 6-5—one need only add the appropriate out-of-loop (OOL) compensation before the wavelet transform in Figure 6-14a and after the inverse wavelet transform in Figure 6-14b.

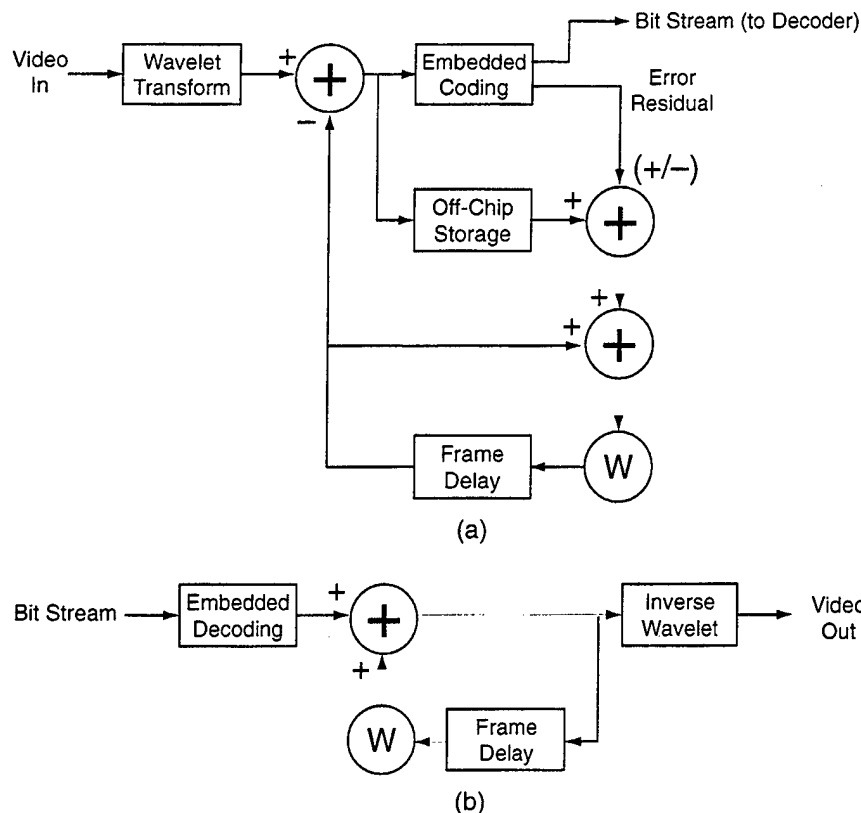


FIGURE 6-19. Robust Video Compression Using Leaky Prediction.  
(a) is encoder and (b) is decoder.

## CONCLUSION

We have presented here a new approach to motion compensation that reduces the complexity of the video encoder by eliminating the inverse transform operation. If integer pixel compensation is desired and the motion vectors are generated by sources external to the encoder (i.e., inertial sensors and gimbal angles), then the complexity reduction achieved by using the new out-of-loop compensation scheme over the conventional hybrid DPCM-transform approaches is 50%. Even with the relatively inefficient (in terms of computational complexity) DFT-based motion estimation algorithms used here, actual complexity reductions of between 19 and 27% have been achieved. Furthermore, the decoder complexity is *not* increased by the proposed motion compensation scheme. In addition, comparisons using a representative video sequence show that the new approach has better rate-distortion performance than the conventional method for a variety of specific implementations. For remotely sensed aerial video, periodic pan compensation appears to be an adequate solution. Nonetheless, this basic methodology still performs as well as or better than the conventional hybrid DPCM-transform approach, even when



used with more sophisticated motion compensation schemes. Finally, we have shown how a video compression system can be made robust to transmission errors using leaky temporal prediction. In summary, out-of-loop motion compensation offers a significant reduction in encoder complexity with no corresponding increase in decoder complexity, while at the same time facilitating spatial scalability and error robustness.

## REFERENCES

- 6-1. C. D. Creusere and Gary Hewer. "Digital Video Compression for Weapons Control and Bomb Damage Indication," *AGARD Conf. Proc.* 576 (September 1995), Chapter 16.
- 6-2. C. D. Creusere. "Periodic Pan Compensation for Reduced Complexity Video Compression," *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing*, Vol. IV (April 1997), pp. 2889-2892.
- 6-3. C. D. Creusere. "A New Approach to Global Motion Compensation Which Reduces Video Encoding Complexity," *Proc. Int. Conf. on Image Processing*, Vol. III (October 1997), pp. 634-637.
- 6-4. K. Challapali and others. "The Grand Alliance System for US HDTV," *Proc. of the IEEE*, Vol. 83, No. 2 (February 1995), pp. 158-174.
- 6-5. D. J. LeGall. "The MPEG Video Compression Algorithm: A Review," *Image Proc. Algorithms and Techniques II*, SPIE Vol. 1452 (1991), pp. 444-457.
- 6-6. Recommendation H.261, "Video Coder for Audio Visual Services at px64 Kbits/sec," The Consultative Committee on International Telephony and Telegraphy (1990).
- 6-7. ITU-T Recommendation H.263, "Video Coding for Low Bitrate Communications" (December 1995).
- 6-8. C. Cafforio and others. "Motion Compensation and Multiresolutional Coding," *Signal Processing: Image Communications*, Vol. 6 (1994), pp. 123-142.
- 6-9. A. Fuldseth and T. A. Ramstad. "Subband Video Coding With Smooth Motion Compensation," *Proc. Int. Conf. on Acoustics, Speech, and Signal Proc.* (1996).
- 6-10. H. Gharavi. "Subband Coding Algorithms for Video Applications: Videophone to HDTV-Conferencing," *IEEE Trans. on Circuits and Systems for Video Tech.*, Vol. 1, No. 2 (June 1991), pp. 174-183.

- 6-11. H.-M. Hang and others. "Digital HDTV Compression Using Parallel Motion-Compensated Transform Coders," *IEEE Trans. on Circuits and Systems for Video Tech.*, Vol. 1, No. 2 (June 1991), pp. 210-221.
- 6-12. J.-R. Ohm. "Three-Dimensional Subband Coding With Motion Compensation," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 559-571.
- 6-13. H. Paek, R. C. Kim, and S. U. Lee. "On the Motion Compensated Transform Coding Technique Employing Sub-Band Decomposition," *Proc. of the SPIE*, Vol. 1818 (VCIP 1992), pp. 253-254.
- 6-14. K. Sawada and T. Kinoshita. "Subband-Based Scalable Coding Schemes With Motion Compensated Prediction," *Proc. of the SPIE*, Vol 2501 (VCIP 1995), pp. 1470-1477.
- 6-15. P. P. Vaidyanathan. *Multirate Systems and Filter Banks*, Englewood Cliffs, NJ, PTR Prentice Hall, 1993, p. 119.
- 6-16. D. Taubman and A. Zakhor. "Multirate 3-D Subband Coding of Video," *IEEE Trans. on Image Proc.*, Vol. 3, No. 5 (September 1994), pp. 572-588.
- 6-17. M. Unser, P. Thevenaz, and L. Yaroslavsky. "Convolution-Based Interpolation for Fast, High-Quality Rotation of Images," *IEEE Trans. on Image Processing*, Vol. 4, No. 10 (October 1995), pp. 1371-1381.
- 6-18. M. J. T. Smith and S. L. Eddins. "Analysis/Synthesis Techniques for Subband Image Coding," *IEEE Trans. Acoust., Speech, Signal Processing*, Vol. 38, No. 8 (August 1990), pp. 1446-1456.
- 6-19. D. Taubman and A. Zakhor. "Orientation Adaptive Subband Coding of Images," *IEEE Trans. on Image Processing*, Vol. 3, No. 4 (July 1994), pp. 421-437.
- 6-20. I. Daubechies. *Ten Lectures on Wavelets*, Philadelphia, PA, SIAM, 1992.
- 6-21. J. Shapiro. "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. on Signal Proc.*, Vol. 41, No. 12 (December 1993), pp. 3445-3462.

## **CHAPTER 7.**

### **REAL-TIME INTRAFRAME ENCODING**

#### **PROCESSOR**

The task of implementing a real-time intraframe encoding of a 512x240 image is performed by a TI DSP, the TMS320C80. This single chip actually contains 5 processors: 4 fixed point Parallel Processors (PPs), and 1 floating point Master Processor (MP). There is a single 64-bit-wide data bus that allows access to the outside world. The Transfer Controller (TC), a peripheral inside of the chip, implements all transactions of data to and from the chip. Fifty kilobytes of memory internal to the TMS320C80 are divided into 25 2-Kbyte RAMs (random access memories). There are three Data RAMs, one Parameter RAM, and one instruction cache per PP. The MP has one Parameter RAM, two Data Cache RAMs, and two Instruction Cache RAMs.

#### **HARDWARE**

We decided to use the Spectrum Signal Processing PCIC80 board running at 40 megahertz. This board provided a single TMS320C80 with 4 megabytes of DRAM, 4 megabytes of SDRAM, 4 megabytes of VRAM for display, a daughter board interface that can accommodate a video acquisition daughter board, and a personal computer (PC) host interface.

#### **TASK PARTITIONING MP VERSUS PP**

Early on we decided that the PPs would not be involved in the data movement and coordination but would yield this task to the MP. The MP would therefore be in charge of using the TC to move data on and off the chip, and also would signal the PPs when data were available. The MP would also handle all of the communication with the PC host. The PPs are well suited to pixel manipulation, and the MP is an excellent overseer. Choosing this partitioning allowed us to use the exact same code on all the PPs.

#### **DEVELOPMENT ENVIRONMENT**

Our current development environment is hosted on the PC platform. A JTAG emulator directly connects to the TMS320C80 to download and debug code. For our demonstrations, we have a PC program running a user interface that downloads and starts the program over the PCI interface to the card, which negates the need for the emulator during demonstrations.

## HIGH LEVEL DESCRIPTION OF ROUTINES USED FOR INTRAFRAME WAVELET CODING

### Master Processor (MP)

Mp.c is the only routine that runs on the MP. This program's main tasks are to start the PPs, to move data to and from the PPs, to coordinate PPs to perform their selected operations, to display raw images and informational text, and to communicate with the PC host. The functions are enumerated in detail here:

1. Copy PP code to SDRAM from DRAM, start PPs executing all the same code.
2. Initialize and kick off the task that will do all the work, and hand over to this task.
3. Set up the video display and acquisition of video.
4. Initialize all the packet transfer tables.
5. Read an image.
6. Read the current compression ratio.
7. Display the image.
8. Perform 3 1/2 level wavelet decomposition on the image.
9. Send the coefficients to the PPs for maximum value calculation.
10. Read all PPs' local maximum values, calculate a global maximum value, and write it back to the PPs.
11. Send groups of six subblocks of the coefficient to each PP and signal them to start.
12. Wait for the resulting bitstream.
13. Write the bitstream to DRAM.
14. Append the dynamic values to the bitstream.
15. Tell the PC host that computation is complete for this frame of video.
16. Display the frame rate and processing time.

## Parallel Processors

Main.c is the routine that initializes all the pointers, coordinates the decomposition of the image, and calls the function to form the bitstream.

Three routines are used to perform the embedded wavelet decomposition: subdec\_vert, subdec\_horiz, and calc\_n\_sub\_mean. The subdec\_vert routine performs the vertical decomposition and the subdec\_horiz routine performs the horizontal decomposition. The calc\_n\_sub\_mean routine performs three functions: it calculates the current image's mean; it subtracts off the previous image's mean from the current image then converts the result to a 16 bit number; and then it shifts the result up 2 places.

Seventeen routines are used to perform the scanning of the embedded zerotrees and to generate the compressed bitstream.

The main routine that coordinates this process is p\_code. It takes care of the maximum value calculation and the calling of all the routines necessary to perform the bitstream encoding.

The zerotree formation is done by input\_block, which takes a block of coefficients in the in-place structure and writes them into another structure in zerotree format. Another routine called comp\_ztr forms a zerotree root structure that aids in scanning of the zerotrees.

The actual scanning of the zerotrees is done in code\_dlist2. Depending upon what information this scanning finds, it either calls comp\_dbits, new\_dbits\_nwo, new\_dbits\_wo, or new\_dbits2. A subordinate pass is run in subpass, which gives an extra refinement to coefficients that have already been chosen as significant.

The arithmetic coder is implemented in start\_model, start\_encoding, start\_outputing, do\_syms, encode\_symbol, bit\_plus\_follow, and update\_model.

The output of the encoder is a series of bits that are packed and put into a structure by new\_output\_bit.

## Speedup Methods

Several methods that are used to speed up the code and improve its efficiency are enumerated below, along with their functions.

### **MP.**

Asynchronous handling of subblocks and bitstream blocks

Frequently used variables stored in internal memory

Predefine Packet transfer tables

Poll for images

Overlap of different PPs' executions

### **PP.** (See Figure 7-1.)

Horizontal and vertical decomposition rewritten in assembly language with parallel instructions

Code written with cache in mind

Caching of symbols

Optimized C code

Subordinate pass rewritten in assembly language

A stored table of significant coefficients, so that searching is not required

Zerotree pruning performed

PP code execution from fast SDRAM

Only internal memory used for variables and stack (no heap)

Fast output\_bit() routine written

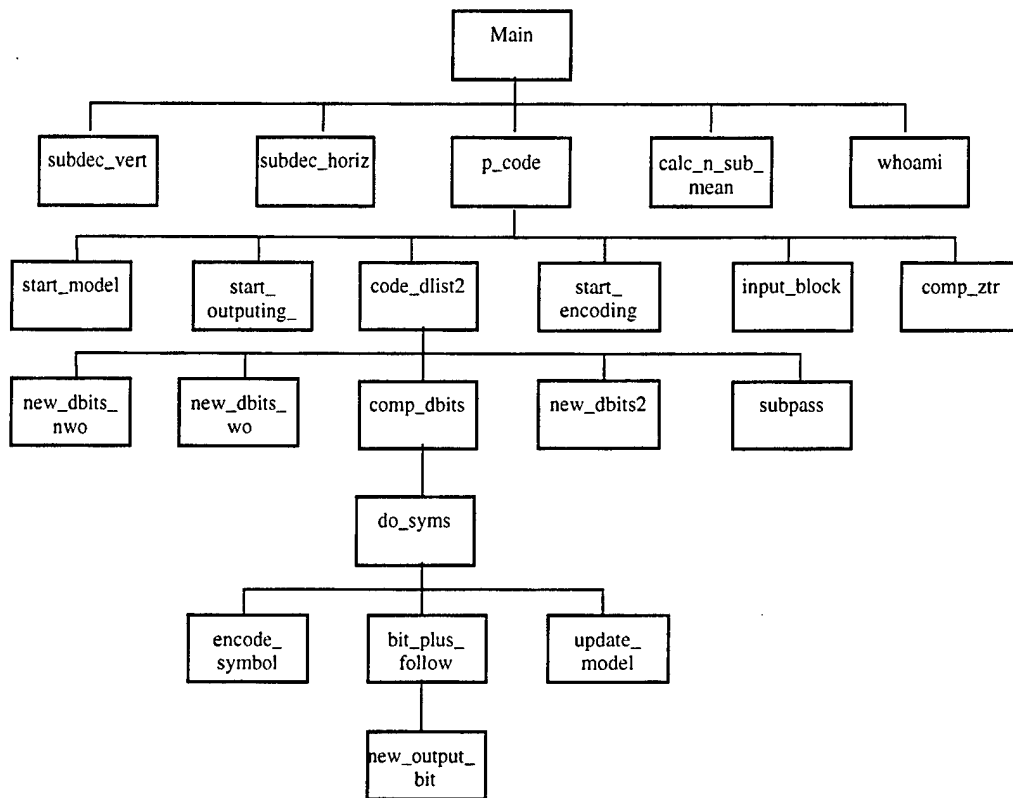


FIGURE 7-1. PP Call Tree.

The files used for this project are:

Master Processor programs	mp.c
Parallel Processor programs	
Include files	arith.h modlp.h
C language files	codenew.c j_enc.c main.c modlp.c
Assembly language files	bitebits.s hvenc.s mean2.s subpass.s ztr.s
Support files	
Batch files for making executables	cc.bat - makes MP code ccpp.bat - makes PP code
Linker command files	pcic80.cmd - linker file for MP link pcic80a.cmd - linker file for PP link



## CHAPTER 8.

### REAL-TIME INTRAFRAME DECODING

The process of decoding is basically the inverse of encoding, where the bitstream is changed back into symbols, a zero-tree is formed, and then the coefficients are inverse transformed into their pixel representation.

#### HIGH LEVEL DESCRIPTION OF ROUTINES USED FOR INTRAFRAME WAVELET DECODING

##### Master Processor (MP)

Mp.c is the only routine that runs on the MP. This program's main tasks are to start the parallel processors (PPs), move data to and from the PPs, coordinate PPs to perform their selected operations, to display images and informational text, and to communicate with the personal computer (PC) host. The functions are enumerated in detail here:

1. Copy PP code to SDRAM from DRAM, start PPs executing all the same code.
2. Initialize and kick off the task that will do all the work, and handover to this task.
3. Set up the video display.
4. Initialize all the packet transfer tables.
5. Wait for a packet to decode.
6. Read the current mean, maximum value, FIRST value, and compression ratio.
7. Send a packet of the bitstream to each PP.
8. Wait for resulting coefficients.
9. Write coefficients to memory.
10. Perform 3 1/2 level wavelet synthesis on the coefficients.
11. Display image.
12. Display frame rate and processing time.

## Parallel Processors

Main.c is the routine that initializes all the pointers, calls the function to decode the bitstream, and coordinates the wavelet synthesis of the image.

Eighteen routines are used to perform the decoding of the bitstream into zero-trees.

The main routine that coordinates this process is p\_dec. It coordinates the input of the bitstream, the initialization of the arithmetic decoder, and the calling of the routine that will decode the bitstream (fast\_arith\_decode2).

Fast\_arith\_decode2 executes one pass through all six of the zero-trees. Each pass can affect single or multiple bitplanes: therefore this routine must be called multiple times until all the input symbols have been decoded. It traverses the zero-trees in parallel from top to bottom, pruning along the way. To turn a symbol into a coefficient it calls new\_dec\_dbits\_ri or new\_dec\_dbits3\_ri, depending on what level of the zero-tree it is parsing. The symbols are pulled from a small cache of symbols that have been decoded so that every time a symbol is needed we do not incur a cache miss to execute the code to fetch a symbol. The cache is small so that we do not waste time caching symbols that will never be used because we change our number of symbols in the arithmetic decoder model. It is impossible to tell *a priori* how many symbols we will need for a pass, because it is dependent upon the symbols that are read in and where they are needed in the tree.

We know how many symbols must be read in only on the last level of the tree (if it has not been pruned off). Pre\_dbits2 takes care of reading in the correct number of symbols for this level.

The pass down function is performed in new\_dec\_dbits\_nri and possibly in dec\_dbits\_fix.

The subband pass is performed in do\_subdec.

The zero-tree to in-place formation is done by form\_img2, which takes a single block of coefficients in the zero-tree structure and writes them into another structure in the in-place format.

The arithmetic decoder is implemented in start\_model, start\_decoding, start\_inputting\_bits, quick\_syms, new\_decode\_symbol, and new\_update\_model.

Bits are read out of the incoming bitstream by new\_input\_bit, one bit at a time.

Three routines are used to perform the embedded wavelet synthesis: subsyn\_vert, subsyn\_horiz, and add\_n\_conv8. The subsyn\_vert routine performs the vertical synthesis, and the subsyn\_horiz routine performs the horizontal synthesis. The add\_n\_conv8 routine performs three functions: it adds the previous image's mean to all

the values, it limits the result to a 10 bit number, and then it shifts the result down 2 places (making it into an 8-bit number).

## **Speedup Methods**

Several methods that are used to speed up the code and improve its efficiency are enumerated below.

### **MP.**

- Asynchronous handling of subblocks and bitstream blocks

- Frequently used variables stored in internal memory

- Predefine Packet transfer tables

- Overlap different PPs' executions

### **PP.** (See Figure 8-1.)

- Horizontal and vertical synthesis rewritten in assembly language with parallel instructions

- Code written with cache in mind

- Caching of symbols

- Optimized C code

- Subband pass rewritten in assembly language

- Zerotree pruning performed

- PP code execution from fast SDRAM

- Only internal memory used for variables and stack (no heap)

- Bit input routine written in assembly language

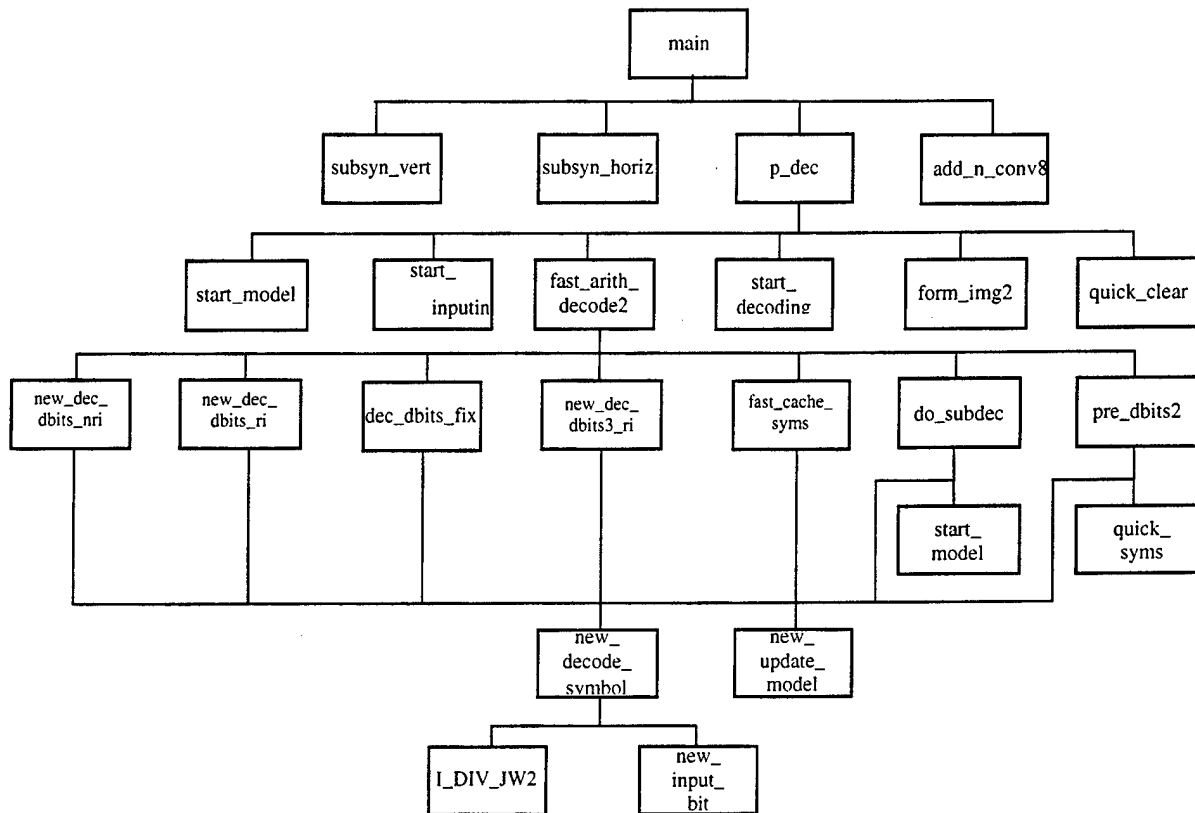


FIGURE 8-1. PP Decode Call Tree.

The files used for this project are:

Master Processor programs	mp.c
Parallel Processor programs	
Include files	arith.h modlp.h
C language files	form.c j_enc.c main.c modlp.c faster_ar.c
Assembly language files	addconv.s decbits.s hvdec.s quick.s sum.s
Support files	
Batch files for making executables	cc.bat - makes MP code ccpp.bat - makes PP code
Linker command files	pcic80.cmd - linker file for MP link pcic80a.cmd - linker file for PP link

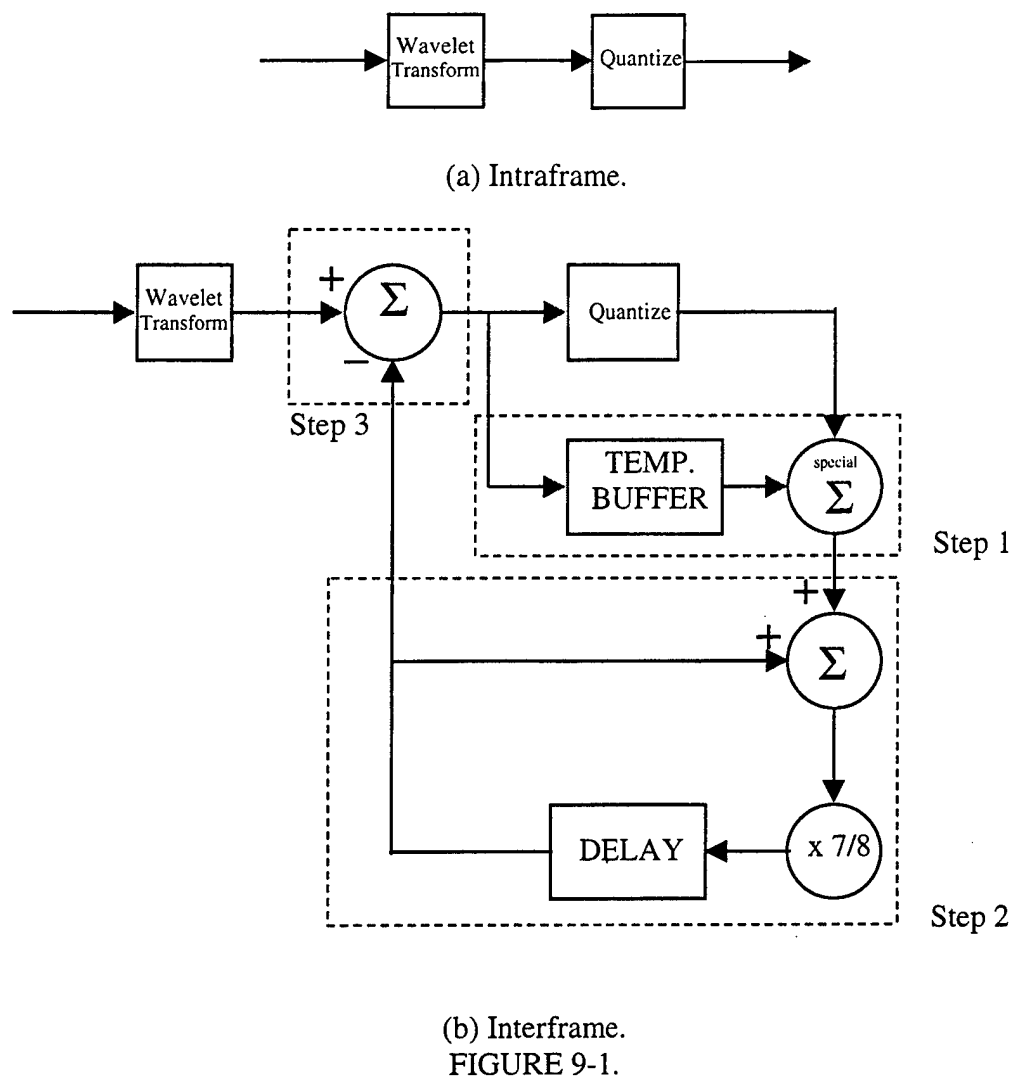
(This page intentionally left blank.)

## CHAPTER 9.

### REAL-TIME INTERFRAME ENCODING

For the purpose of being concise, only the differences between the Interframe and Intraframe methods will be discussed for both the encoding and decoding methods. It can be assumed that the other routines have not changed, and still perform the same functions.

Figure 9-1 is a block diagram outlining the differences between the Intraframe and the Interframe methods of encoding. The Interframe method adds a feedforward step and a feedback step to the process.



The differences will be outlined by the three different steps that have been highlighted in the diagram. Step 1 is used to determine the coefficients that have been received by the decoder. This method is more efficient than doing the inverse quantization and inverse wavelet transform to determine what coefficients were coded. The diff2 routine performs the special summation. This special summation was required to correctly determine the actual value that was sent to the decoder.

Step 2 is our feedback loop to "leak" out the energy from the coefficients that were already sent. The routine that implements this loop is the sum1 routine.

Step 3 is where the new coefficients are compared with a delayed version of what has already been coded and sent to the decoder. This is implemented in the diff1 routine.

Figure 9-2 shows the PP encode call tree.

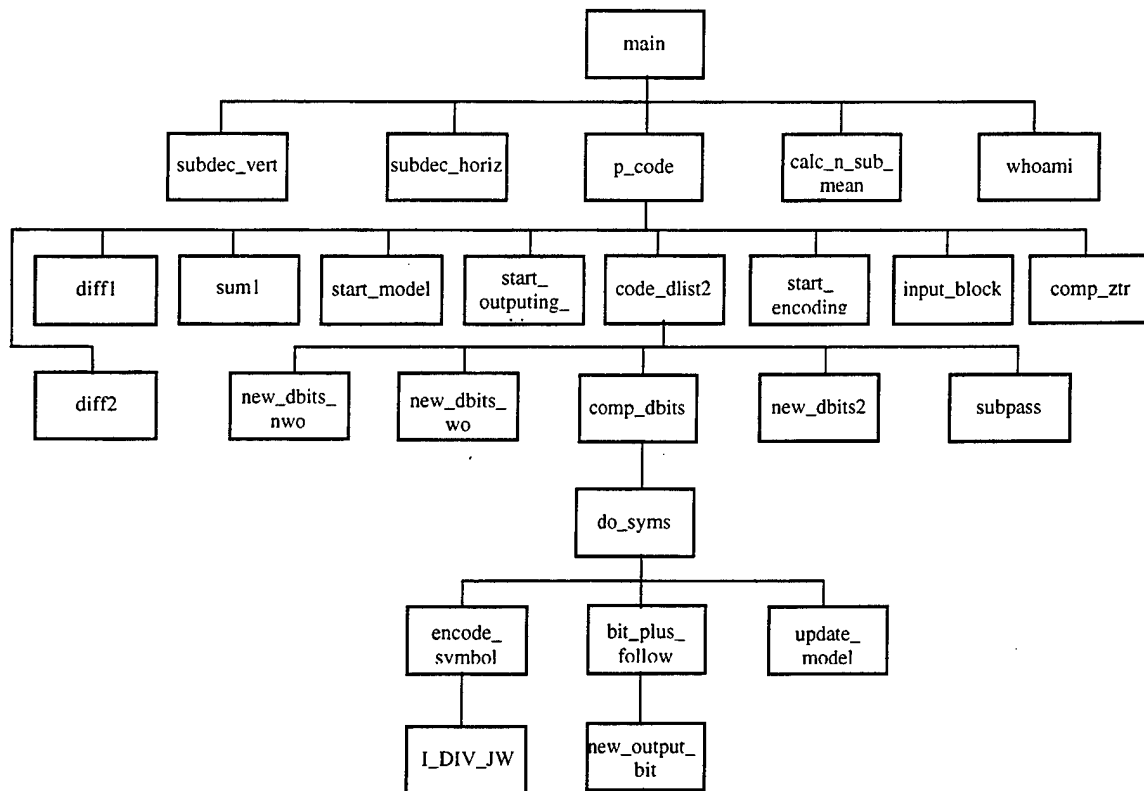


FIGURE 9-2. PP Encode Call Tree.



## CHAPTER 10.

### USERS MANUAL FOR REAL-TIME SOFTWARE

This section describes how to run the real-time software on the PC host platform.

The software on the PC is set up in a client/server relationship. The software that runs on the encode PC is the server, and the software that runs on the decode PC is the client. The client initiates requests to the server for a frame of encoded data and designates which program will run at the beginning of a connection.

#### ENCODE

The name of the software on the encode side is Gserv. This software needs no user interface: it was meant to be started and then left with only the client controlling it. There are no buttons or pull downs on its menu. Figure 10-1 shows the window for this program.

To use the Gserv program, simply double click on the Gserv icon on the desktop. A camera must be attached to the red input of the PCIC80 video input card, and optionally a multisync monitor may be attached to the PCIC80 display output. No images will display on the monitor until the Gclient program (running on the decode PC) is invoked and the Connect button is pushed.

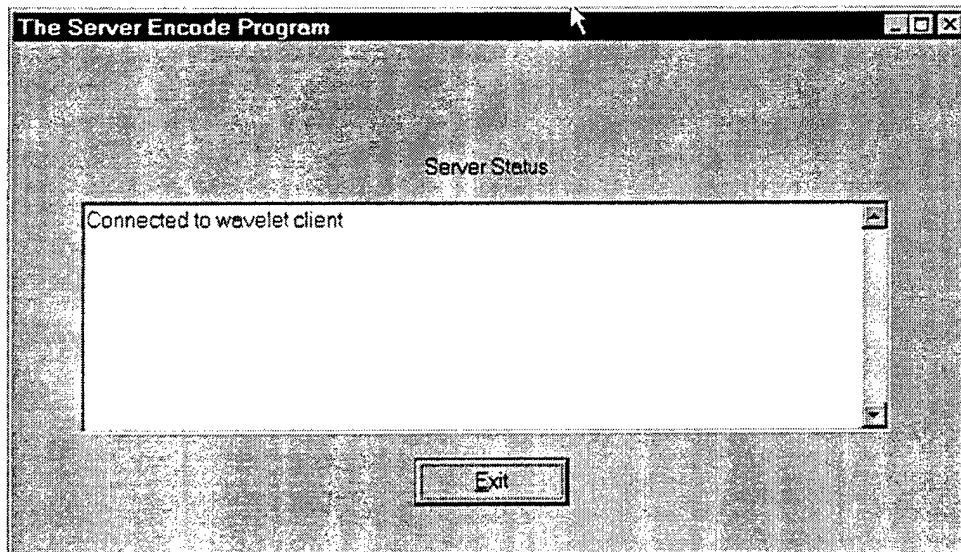


FIGURE 10-1. Gserv Window for Encode.

## DECODE

The name of the software on the decode side is Gclient. This software has a fairly detailed user interface: the user can select from four different versions of C80 code, select one of five different compression ratios, throttle the bandwidth of the communication channel (slow the encoder down), and select the server's IP address. Statistics of the software in the form of frame rate, frame transfer time, number of total frames transferred, plus the number of lost frames are all being displayed. Figure 10-2 shows the window for this program.

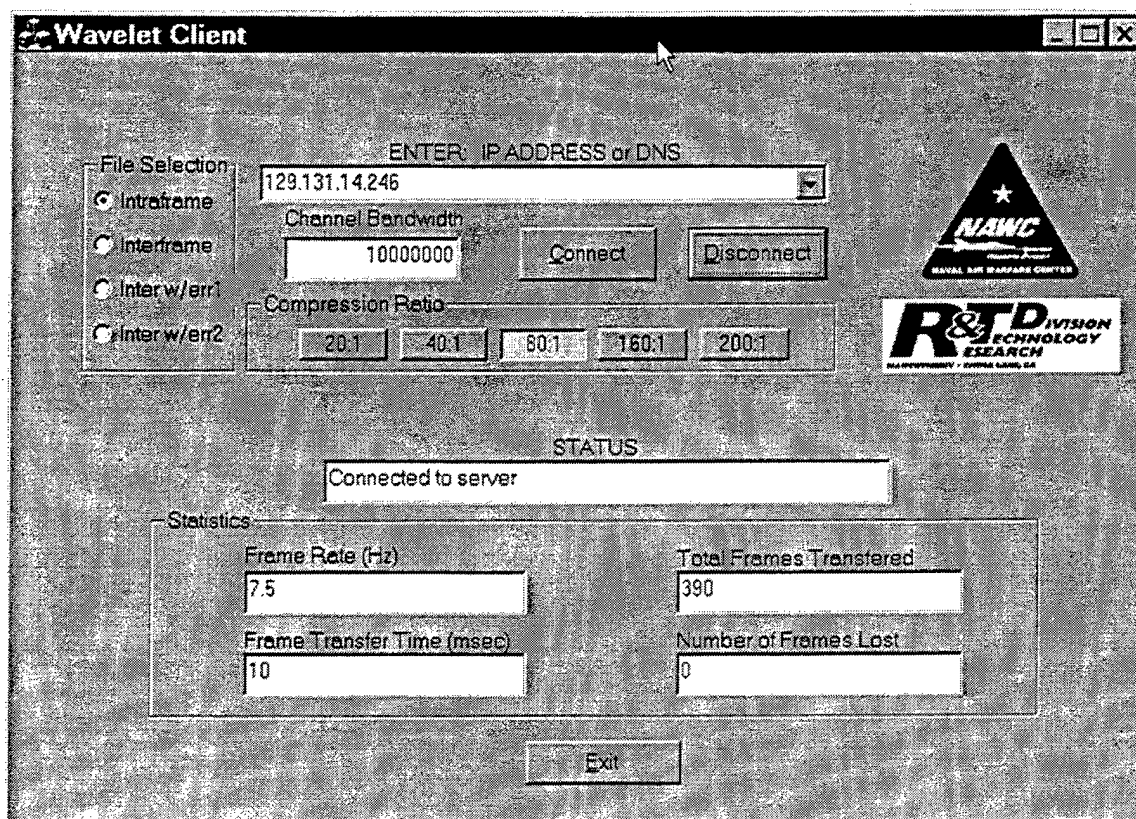


FIGURE 10-2. Gclient Window for Decode.

To use the Gclient program, simply double click on the Gclient icon on the desktop. You must then select the IP address of the encoder PC, either from the list or by typing it. You then choose which file you would like to run (different files are run on both the encoder and the decoder). You may choose the compression ratio that you desire and then click Connect. You cannot change the compression ratio or the file selection while you are connected: you must first disconnect, then make your choice of file or compression ratio, then connect again.

## CHAPTER 11.

### CONCLUSIONS AND FUTURE WORK

In the course of this project, we have carefully studied the application of video compression to the military remote sensing problem. After considering a variety of algorithms, we ultimately concluded that a wavelet-based embedded coding scheme was the best choice for these military applications because of its ability to produce progressively decodable, compressed representations of images while still achieving excellent rate-distortion performance (i.e., minimal distortion for a given compression ratio). Starting with the EZW compression algorithm, we first modified it to add robustness to transmission errors. This robustness allows our algorithm to gracefully degrade in the presence of uncorrected transmission errors—a feature not shared by existing standardized algorithms such as MPEG and JPEG. To overcome the major drawback of embedded coding—its slow execution speeds—we have developed parallel implementations and also introduced the algorithmic concepts of adaptive embedding and cache-based zerotree processing. From our studies on parallelization, we determined the optimal parallelization of the wavelet transform for an array of parallel processors with given communication and execution speeds. We have also developed and applied for a patent on parallel SIMD implementations of an EZW encoder and decoder. Our work on algorithmic speedups led to the development of adaptive embedding and cache-based zerotree processing. These topics are not covered in this technical publication because of space considerations, but a paper on them is scheduled to appear in the December 1999 *IEEE Transactions on Image Processing*. The final cornerstone of our real-time embedded implementation is the concept of entirely in-place wavelet coefficient calculation and coding, for which a patent application has been submitted. The use of in-place transformation and coding greatly reduces the amount of memory required to perform embedded encoding and decoding without negatively affecting the rate-distortion performance of the complete system.

Because the major focus of our research has been to develop compression algorithms that function well in the real world (with its unstable, error prone communications channels and strict space and power constraints), we felt that it was important to actually demonstrate our algorithms using real-time hardware. To accomplish this, we ported our algorithms to the TMS320C80 processor and developed a Transmission Control Protocol/Internet Protocol (TCP/IP) packetization system to send real-time video across the Internet. While transmission of compressed video via a military radio would have made for a better demonstration, we did not have access to such equipment nor did we have the funds to purchase such equipment. The Internet demonstration is nonetheless valuable because ATM transmission channels such as the Internet represent the future of communications. Our real-time demonstration system uses adaptive embedding, cache-based zerotree processing, and in-place coding to achieve high speed throughput with minimal memory requirements. It also uses our robust EZW intraframe compression

along with "leaky" interframe predictive coding. Furthermore, the software is configured so that the operator viewing the decoded video can adjust the compression ratio and turn features (i.e., interframe compression) on or off as desired.

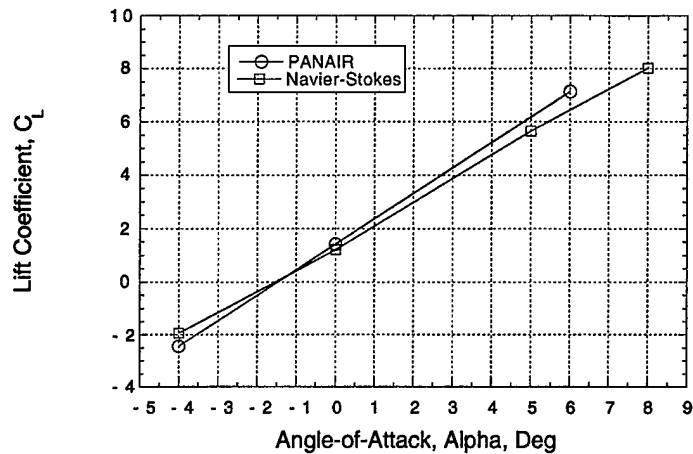
To evaluate the success of an Office of Naval Research (ONR) funded research project in today's environment, one must look at technology transitions. The Data Compression Project has been extremely successful in this regard—the next generation tactical Tomahawk has selected our compression algorithm for its baseline BDI transmission system. Because the demands of their application are not severe (they are transmitting only 1 image over a slow SATCOM link), they have selected a fairly basic version of our algorithm—EZW compression with in-place coefficient calculation and coding—for decreased memory usage and more predictable execution speed. Another potential transition we have pursued is the Rockwell/Collins Surgical Strike datalink program. To this end, we have ported our algorithm to the same dual TMS320C80 processor card used in their program along with an identical video capture daughterboard. Thus, by just loading our software into their system, we should be able to test our algorithm with their communications link. Because of Surgical Strike's budget constraints, however, we have not as yet been able to try out our compression algorithms with their system. We hope that an opportunity will arise at some future date.

Despite our many successful accomplishments during the last 4 years, there are a number of things that we could not do because of time limitations and financial constraints. For example, we were unable to add out-of-loop motion compensation (discussed in Chapter 6) to our real-time demonstration software. Such compensation can dramatically improve compression performance in scenes with large camera or platform motions. We also did not have the opportunity to optimize our combined REZW/leaky predictive video coding algorithm for different packet and/or bit error rates. The tables generated by this optimization could then be used to dynamically control leakage and partitioning parameters to guarantee the best possible decoded image, given the current channel noise state. Other topics we hope to address in our future research include real-time region-of-interest compression, list Viterbi convolutional decoders for even better error resilience, and embedded algorithms that are optimized to run on VLIW (very long instruction word) processors such as TI's C6X series.

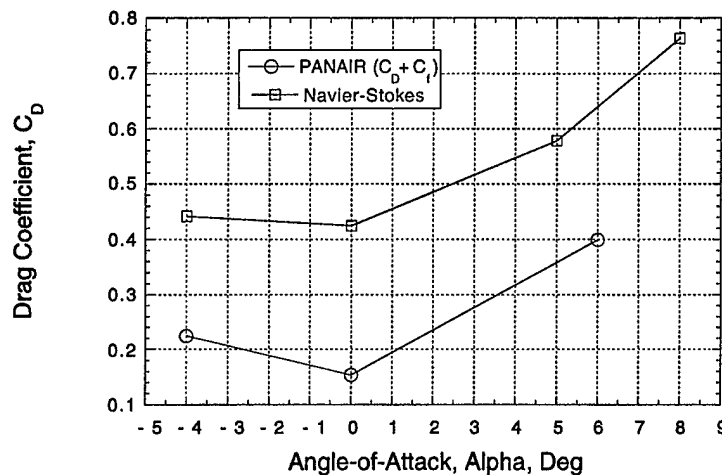
In summary, we are very pleased with the results that we have achieved in the course of this project. We have made significant scientific contributions to the state-of-the-art in embedded image compression, and we have also fielded a system that demonstrates the real-time capabilities of our research results. Probably more important than our four journal papers and two patents, however, is the fact that we have transitioned technology developed here with ONR 6.2 funding directly into a weapons system. Although there is more work left to do in this area, we believe that, by virtually any measure, this project has been highly successful. We are very satisfied with what we have accomplished.

## APPENDIX A

### AERIAL PLATFORM ANALYSIS

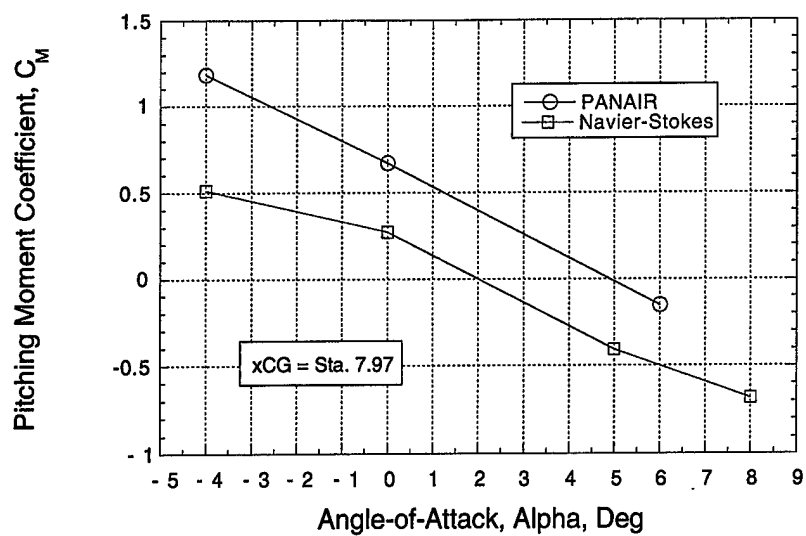


(a) Lift coefficient comparison of PANAIR solutions and Navier-Stokes solutions at sea level with turbulent boundary layer.

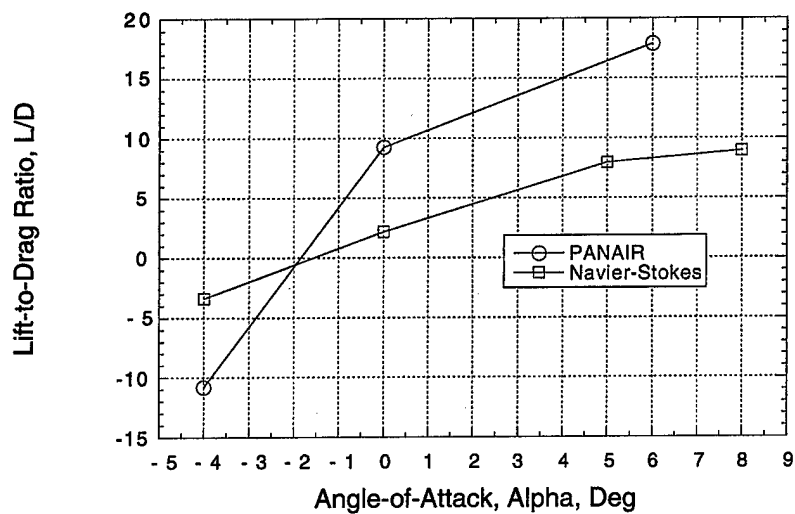


(b) Drag coefficient comparison of PANAIR solutions and Navier-Stokes solutions at sea level with turbulent boundary layer.

FIGURE A-1. Force and Moment Comparisons of the PANAIR, With Additional Viscous Drag From VSAERO, and Navier-Stokes Solutions. The solutions are for Mach 0.09 at sea level. All solutions presented in this appendix are without vertical fins or actuator disk.

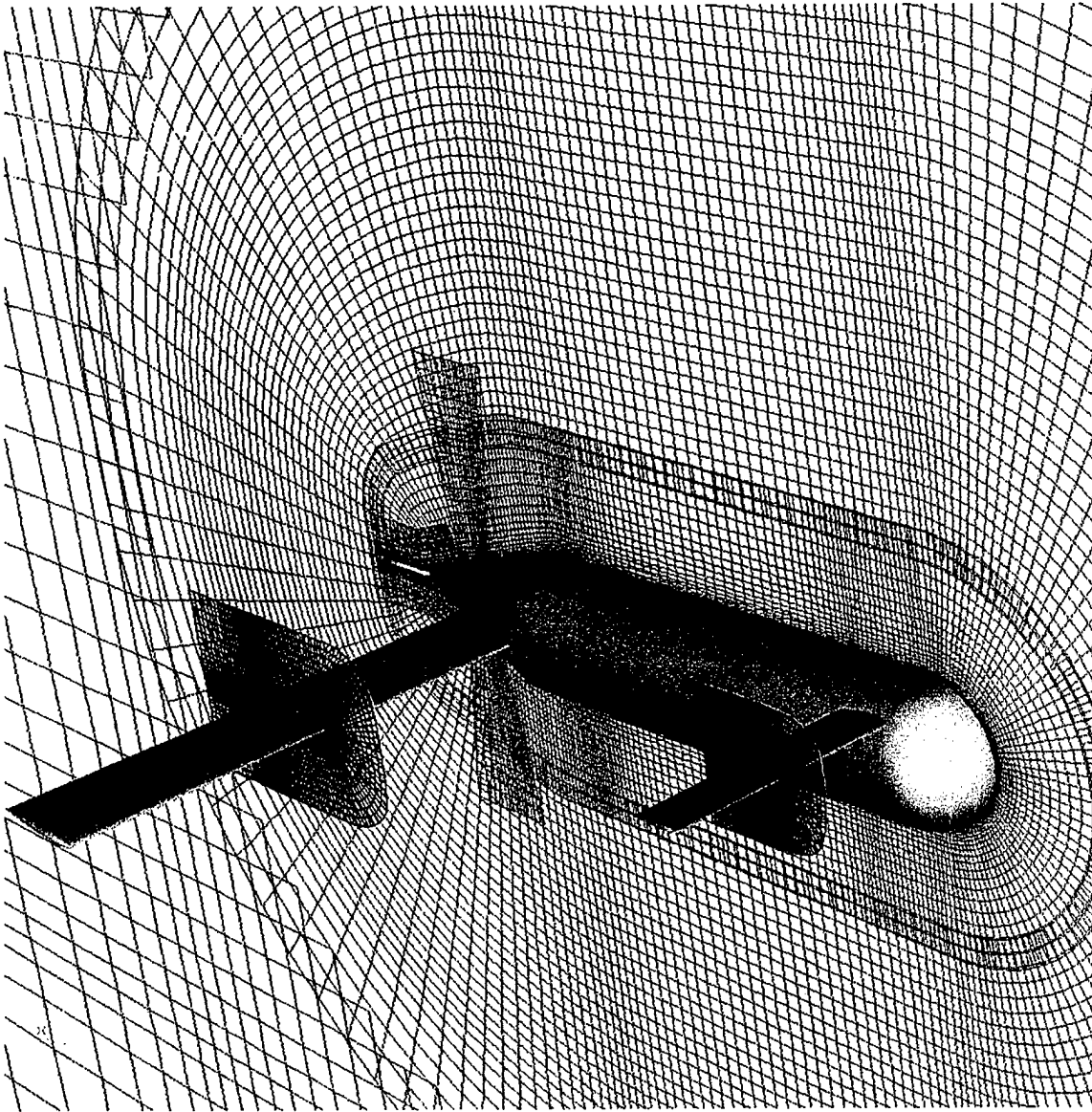


(c) Pitching moment coefficient comparison of PANAIR solutions and Navier-Stokes solutions at sea level with turbulent boundary layer.



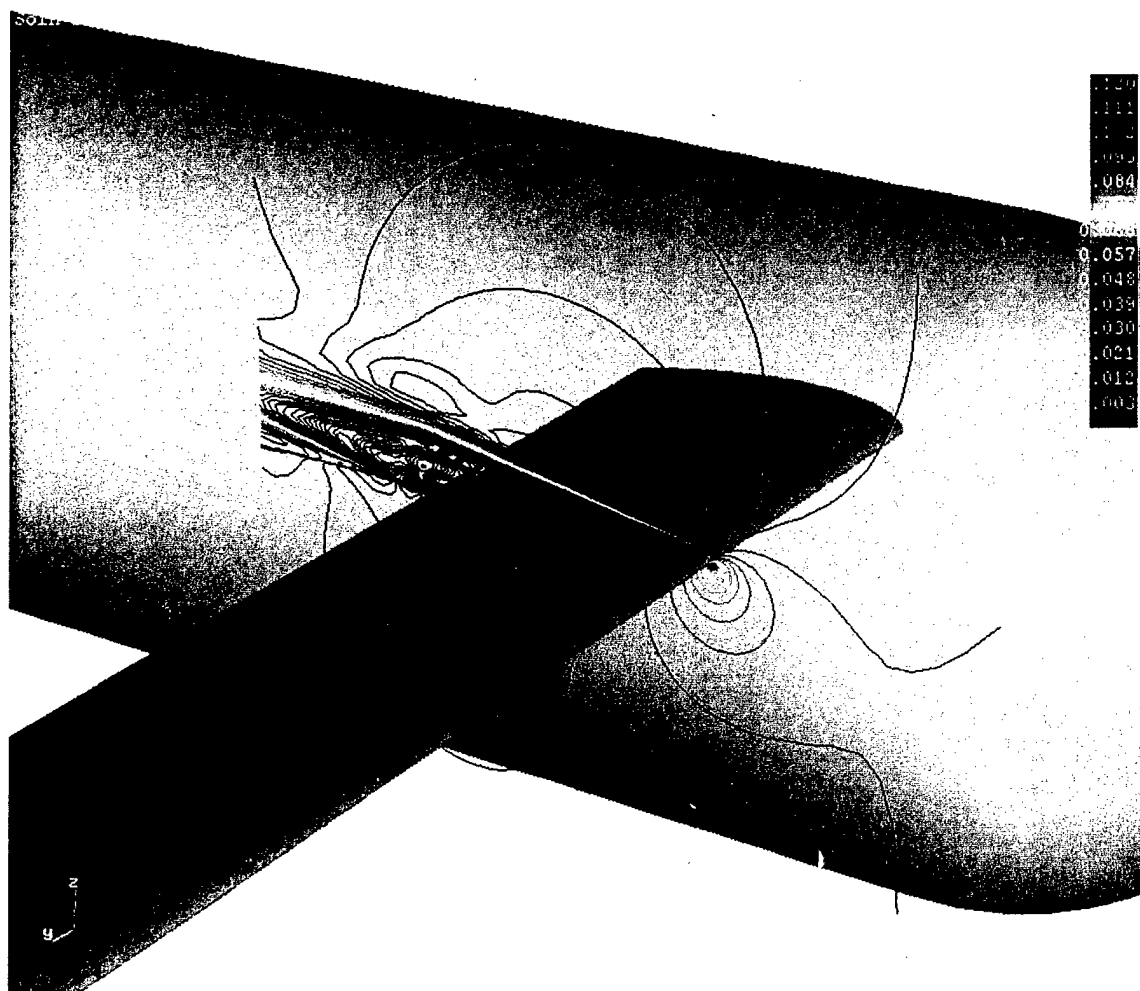
(d) Lift-to-drag ratio comparison of PANAIR solutions and Navier-Stokes solutions at sea level with turbulent boundary layer.

FIGURE A-1. (Contd.)

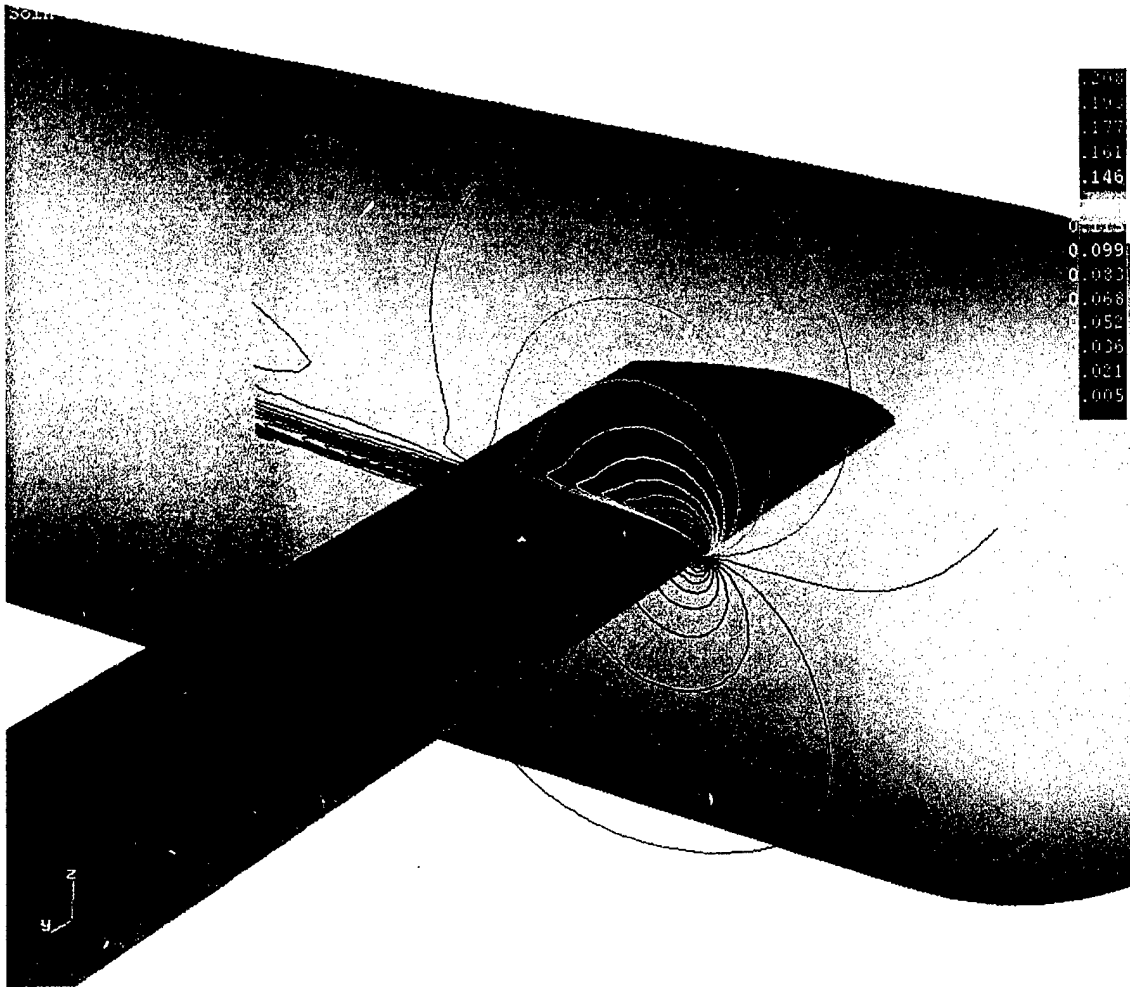


(a) The grid point distributions in the symmetry plane and about the canard and wing.

FIGURE A-2. The Grid Point Distributions in the Symmetry Plane and About the Canard and Wing. Mach contours on the upper surface of the canard for laminar and turbulent boundary layer specifications at 0-degree angle of attack and sea level altitude.

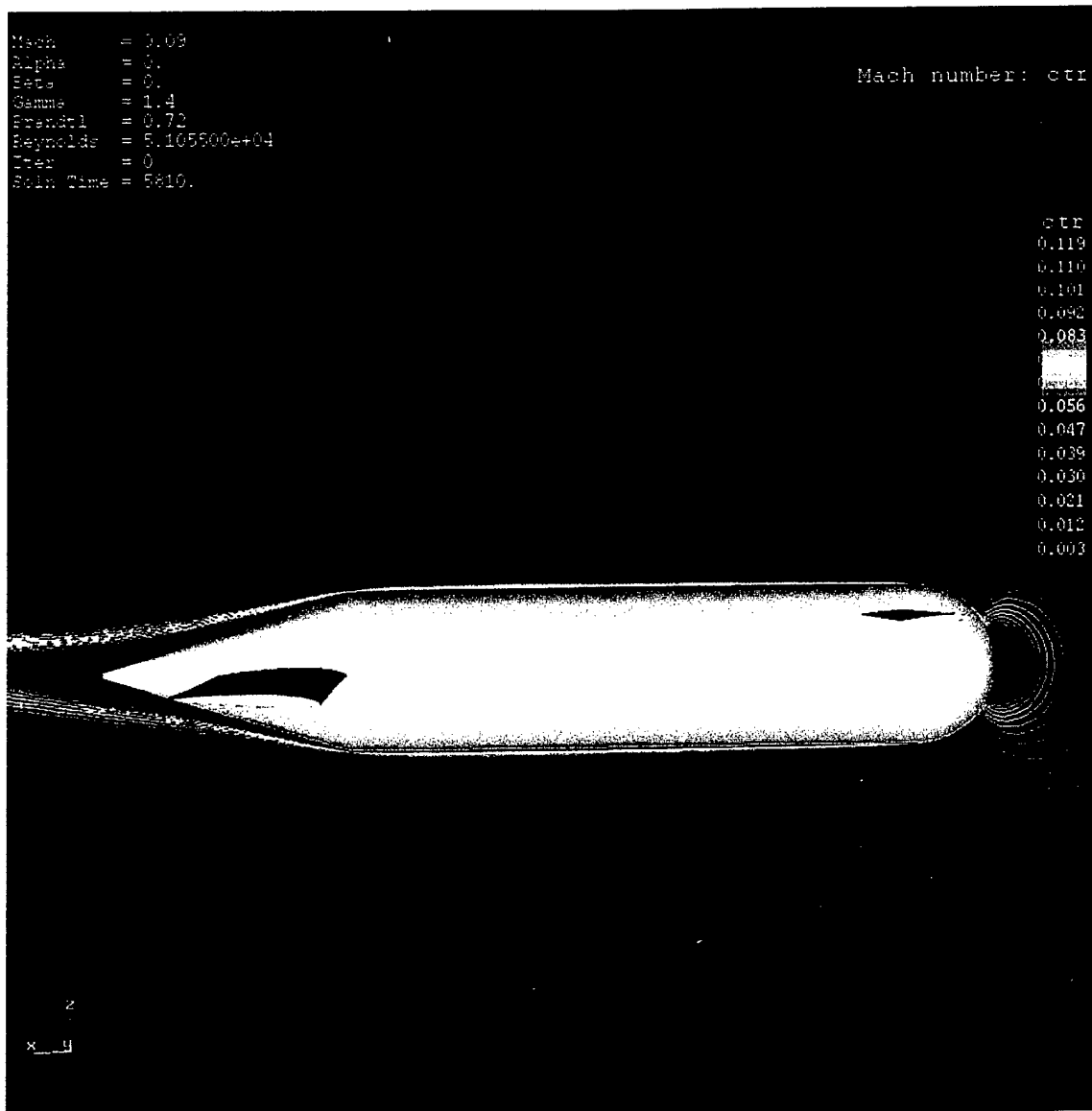






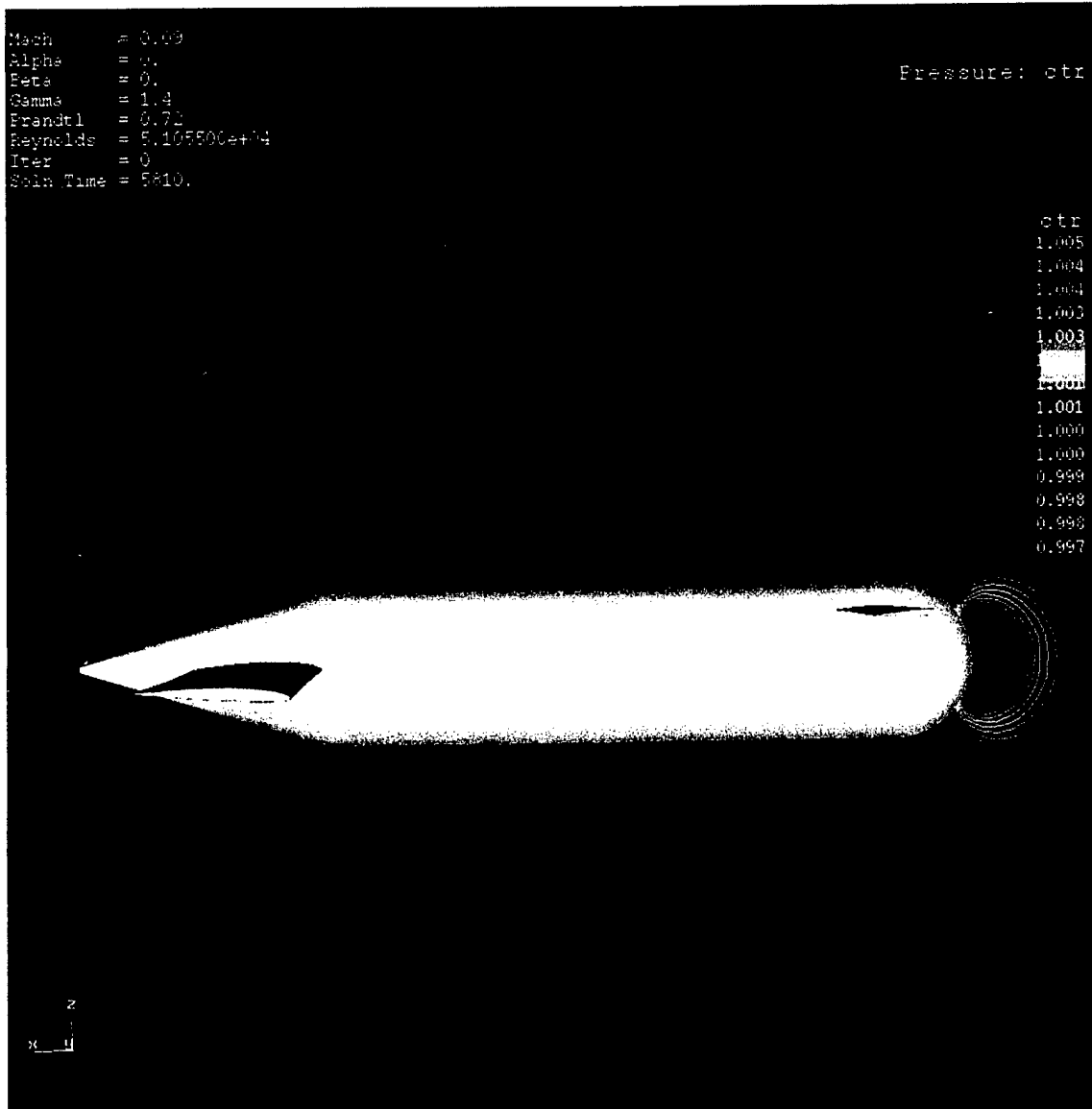
(c) Turbulent boundary layer, Mach contour distribution about the canard.

FIGURE A-2. (Contd.)



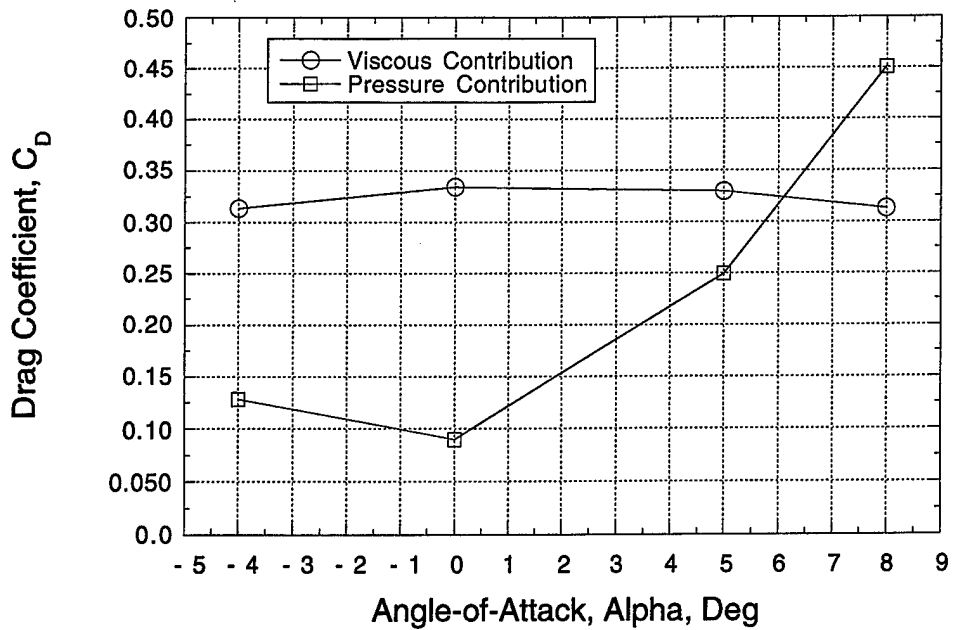
(a) Turbulent boundary layer, Mach contour at the plane of symmetry.

FIGURE A-3. Mach and Pressure Contours at the Plane of Symmetry for 0-Degree Angle of Attack at Sea Level With a Turbulent Boundary Layer.

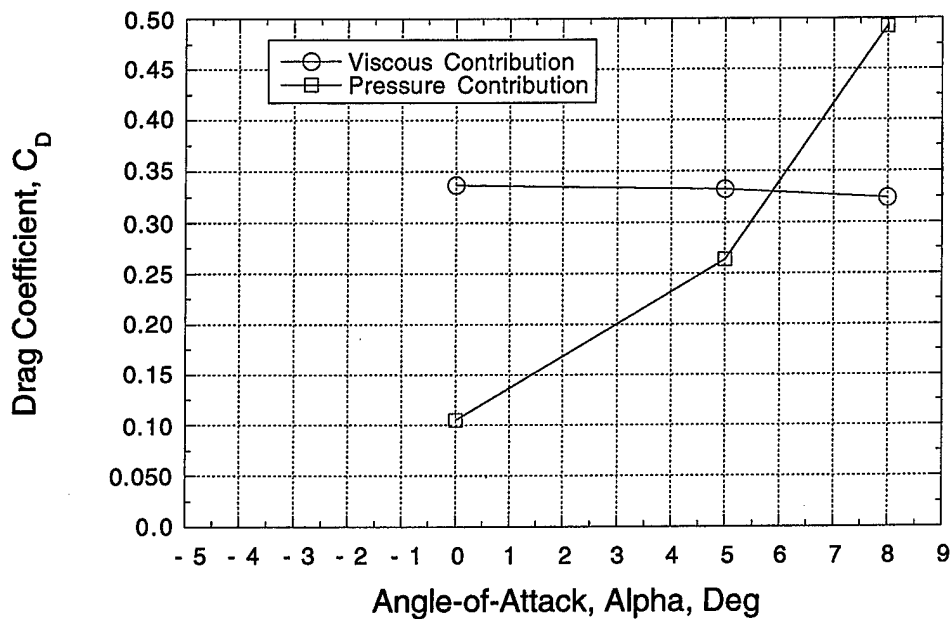


(b) Turbulent boundary layer, pressure contour at the plane of symmetry.

FIGURE A-3. (Contd.)

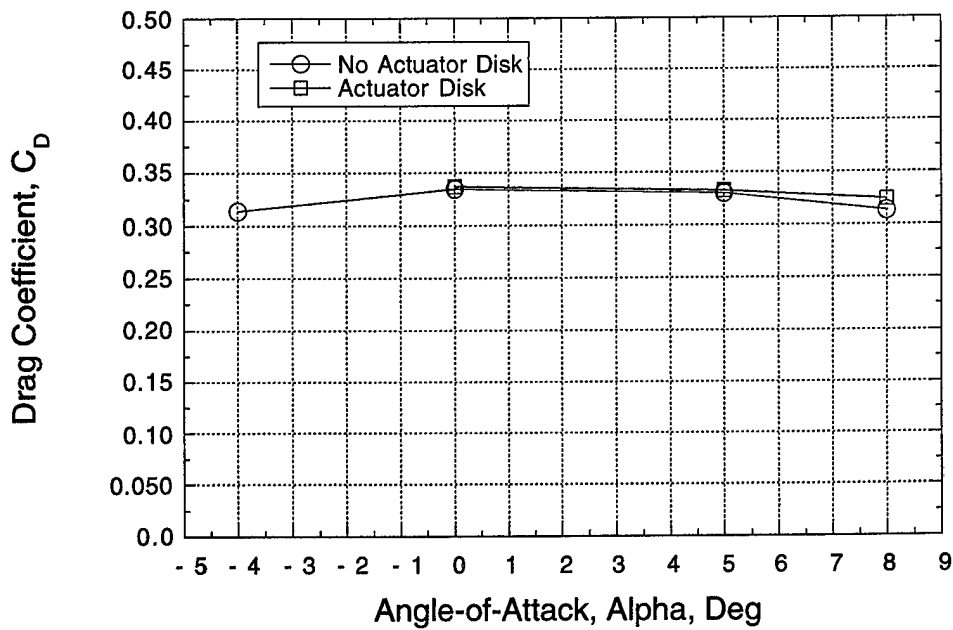


(a) Viscous and pressure contributions to drag with a turbulent boundary layer at sea level, without the actuator disk.

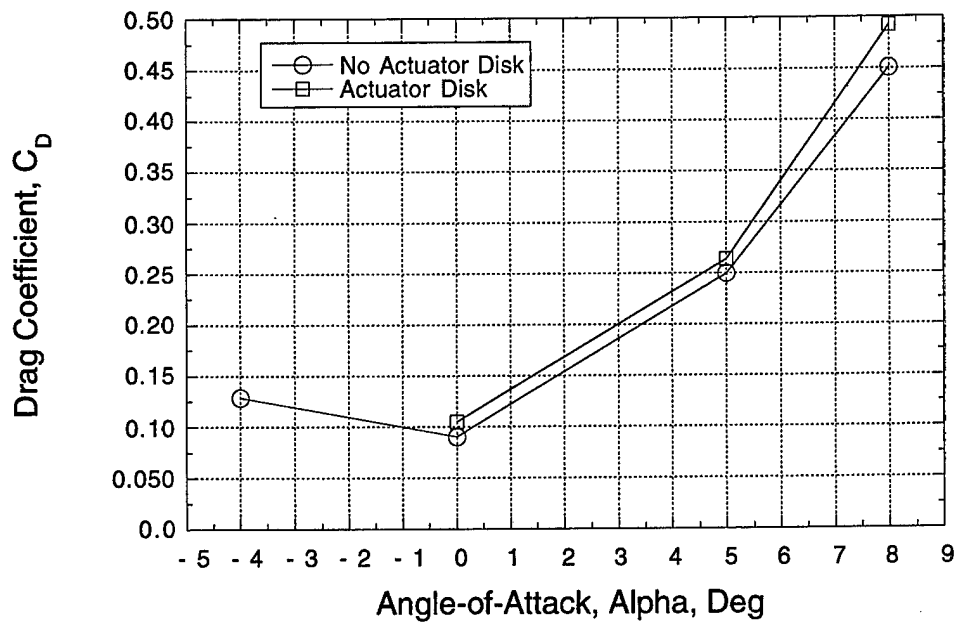


(b) Viscous and pressure contributions to drag with a turbulent boundary layer at sea level, with the actuator disk.

FIGURE A-4. Pressure and Viscous Contributions to the Drag Coefficient With and Without the Propeller.

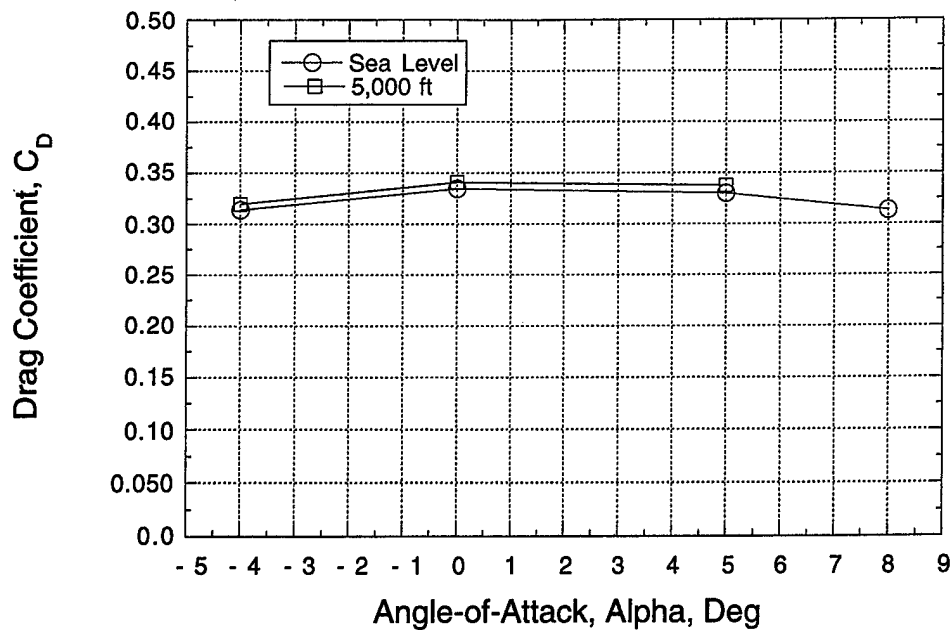


(c) Viscous drag contributions with a turbulent boundary layer at sea level, with and without the actuator disk.

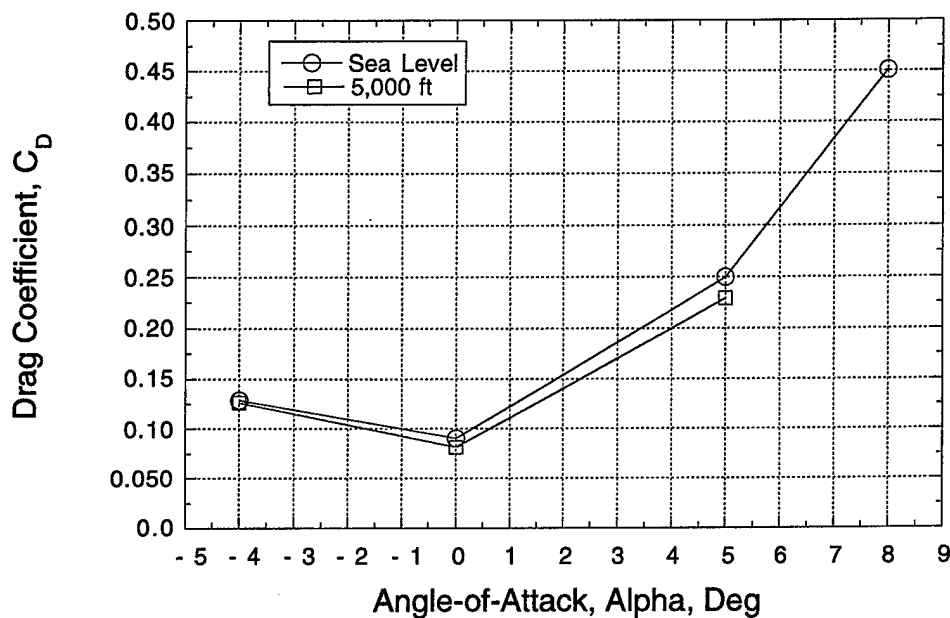


(d) Pressure drag contributions with a turbulent boundary layer at sea level, with and without the actuator disk.

FIGURE A-4 (Contd.)

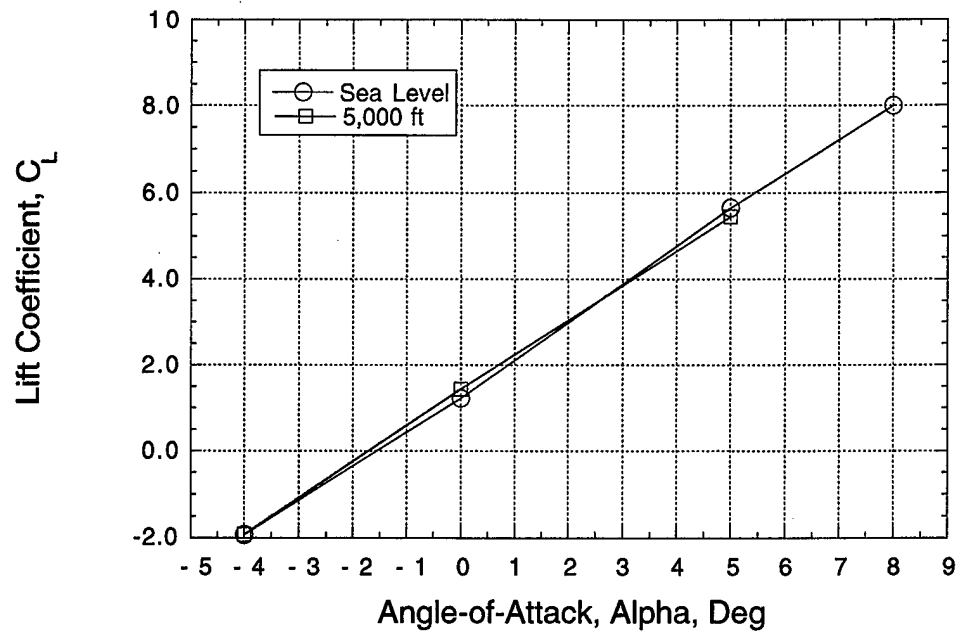


(e) Viscous drag contributions with a turbulent boundary layer at sea level and 5,000-foot altitude, without the actuator disk.

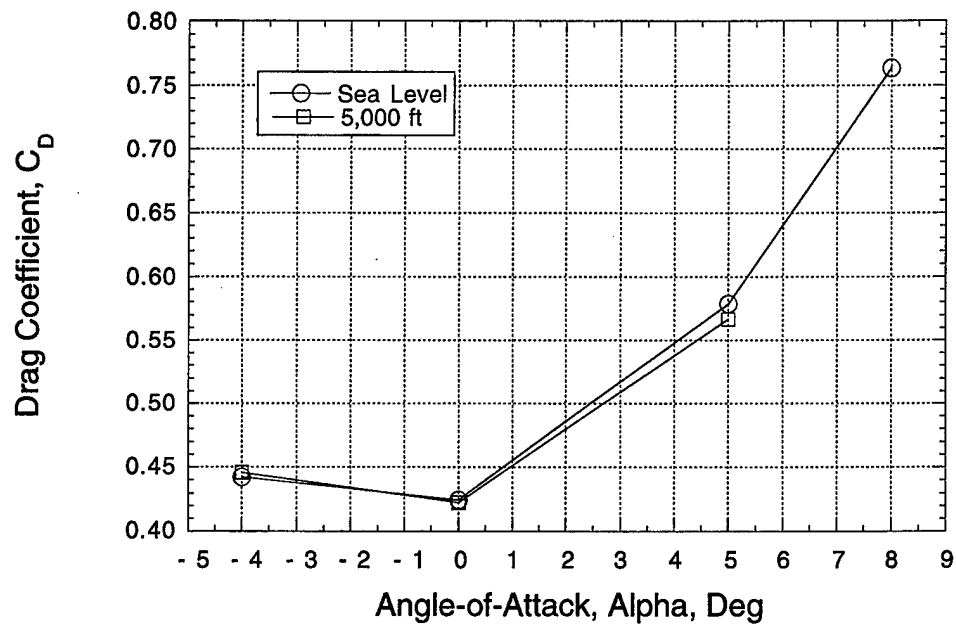


(f) Pressure drag contributions with a turbulent boundary layer at sea level and 5,000-foot altitude, without the actuator disk.

FIGURE A-4 (Contd.)

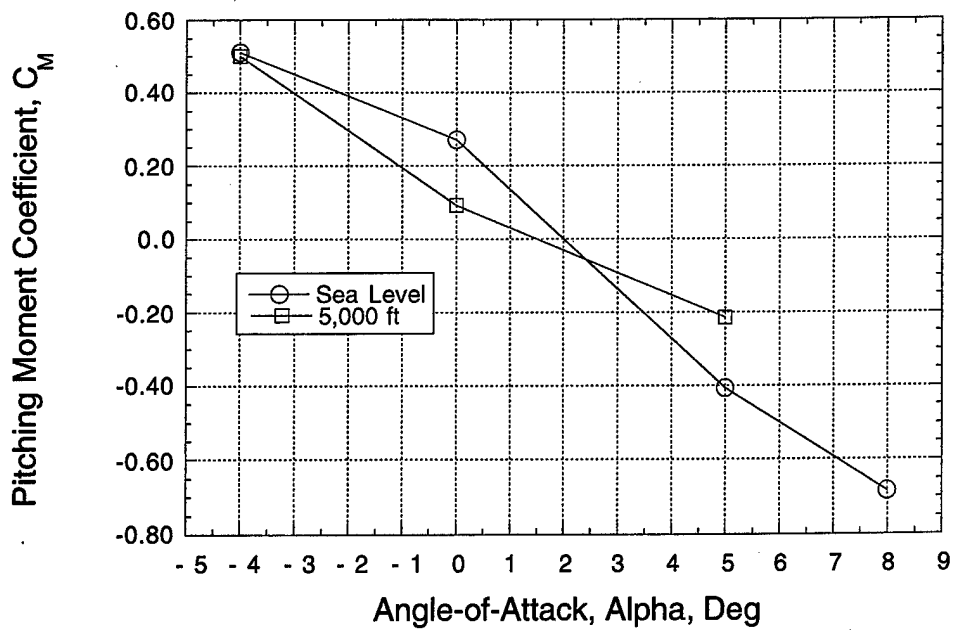


(a) Lift coefficient at sea level and 5,000-foot altitude.

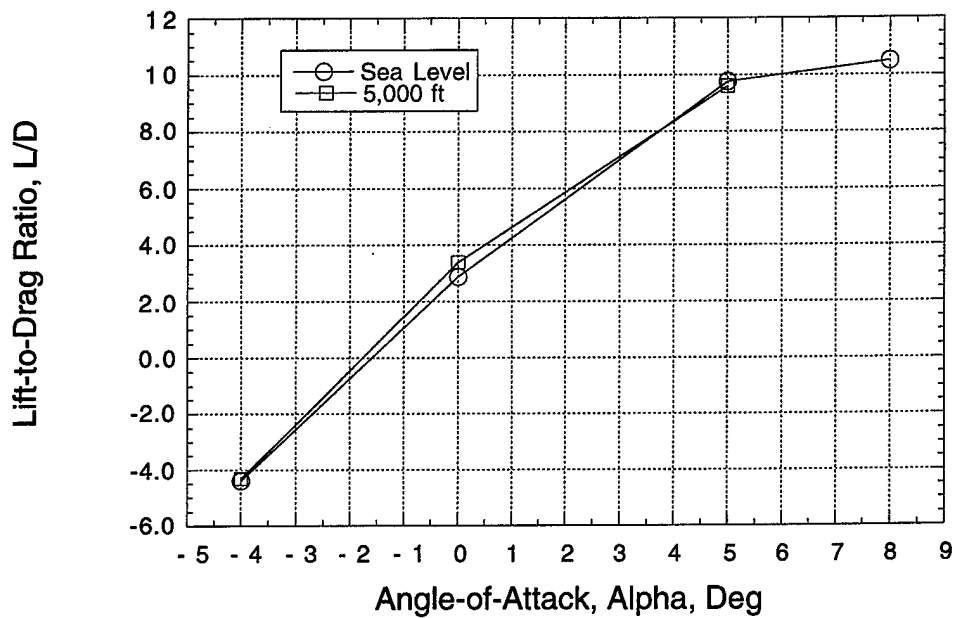


(b) Drag coefficient at sea level and 5,000-foot altitude.

FIGURE A-5. Force and Moment Solution Comparisons for Sea Level and 5,000-Foot Altitudes.



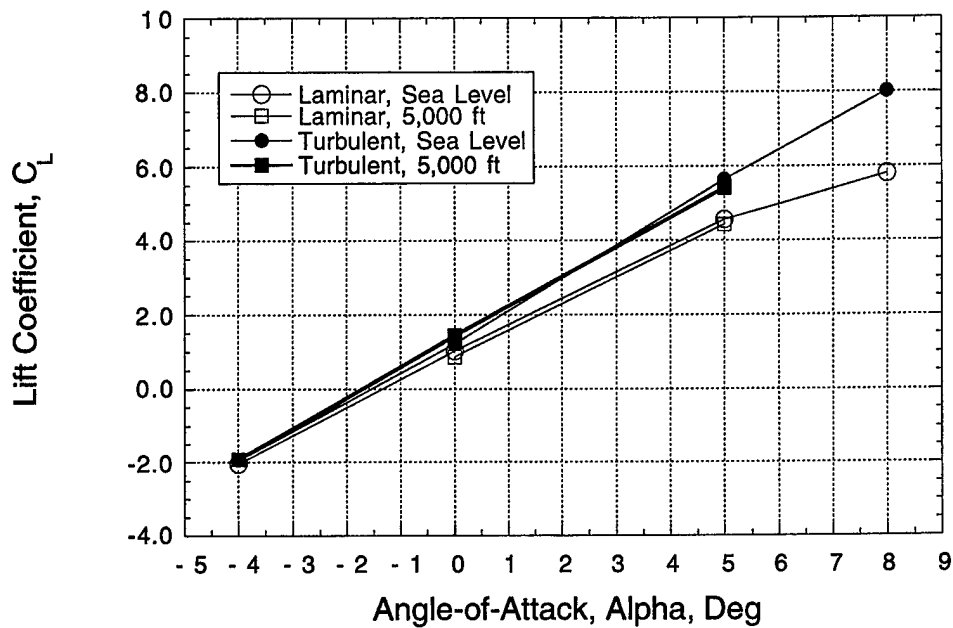
(c) Pitching moment coefficient at sea level and 5,000-foot altitude.



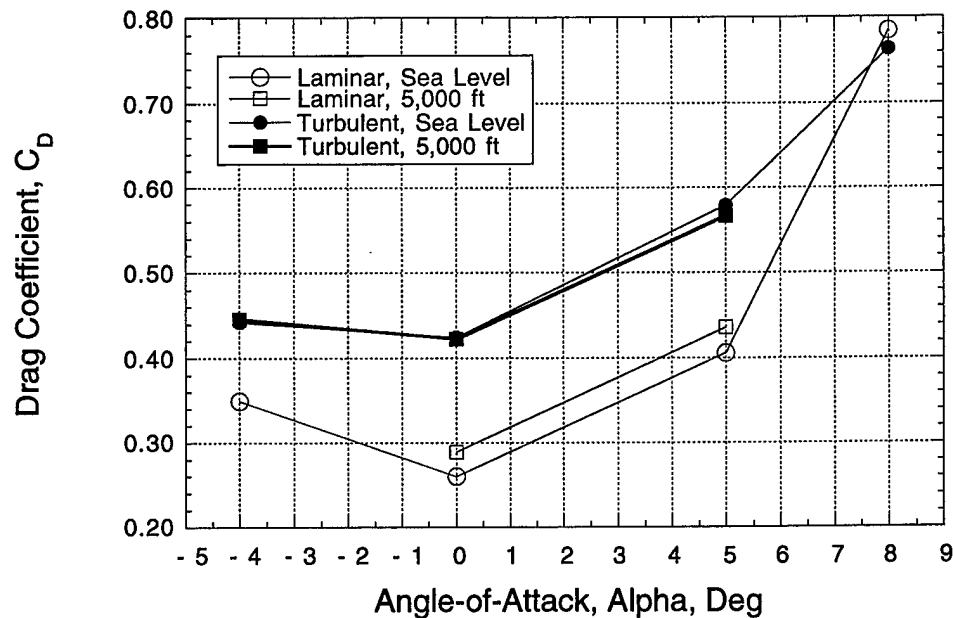
(d) Lift-to-drag ratio at sea level and 5,000-foot altitude.

FIGURE A-5. (Contd.)



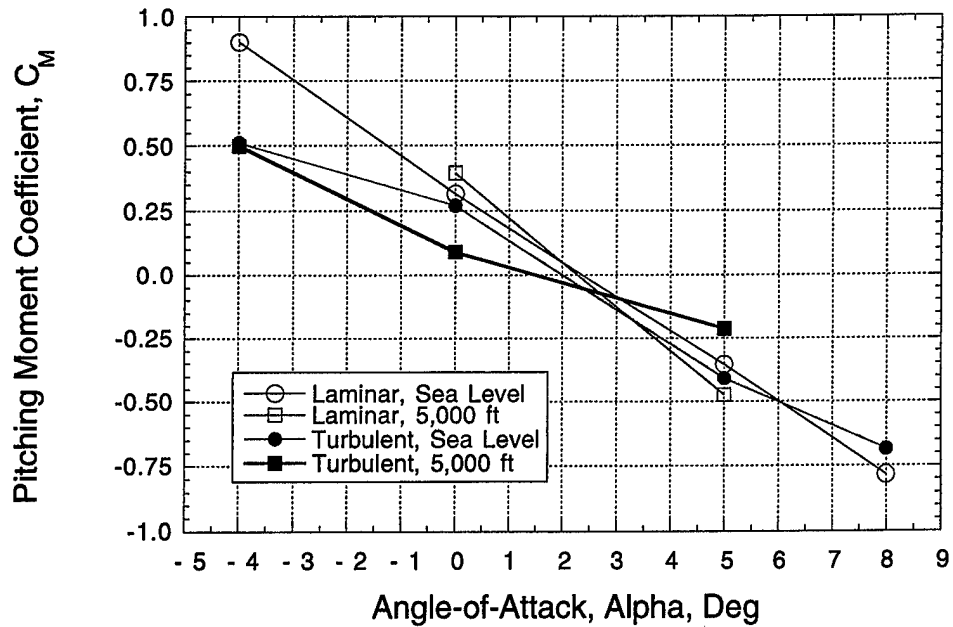


(a) Lift coefficient with laminar and turbulent boundary layers.

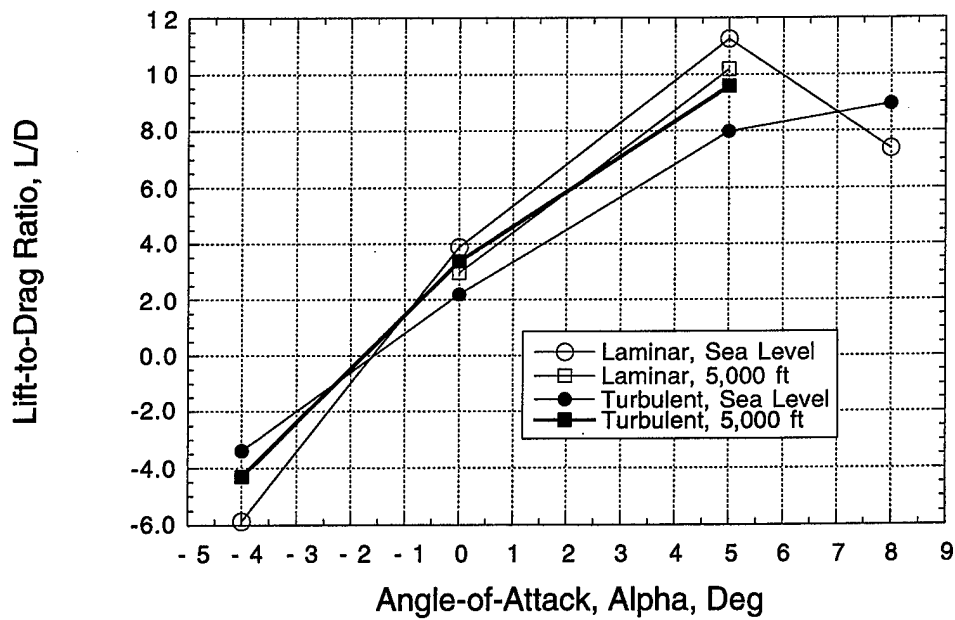


(b) Drag coefficient with laminar and turbulent boundary layers.

FIGURE A-6. Comparison of Forces and Moments With Laminar and Turbulent Boundary Layers at Sea Level and 5,000-Foot Altitudes.



(c) Pitching moment coefficient with laminar and turbulent boundary layers.



(d) Lift-to-drag ratio with laminar and turbulent boundary layers.

FIGURE A-6. (Contd.)

## **APPENDIX B**

### **SOURCE CODE LISTING FOR REAL-TIME COMPRESSION/DECOMPRESSION SYSTEM**

This page intentionally left blank.

## NAWCWD TP 8442

```
/* DECLARATIONS FOR ARITHMETIC CODING AND DECODING */

/* Size of arithmetic code values */

#define Code_value_bits 9 /* Number of bits in a code value */
typedef short code_value; /* Type of arithmetic code value */

#define Top_value (((long)1<<Code_value_bits)-1) /* Largest code val */

/* Half and Quarter points in code value range */

#define First_qtr (Top_value/4+1) /* Points after first quarter */
#define Half      (2*First_qtr)   /* Points after first half */
#define Third_qtr (3*First_qtr)   /* Points after third quarter */
```

```

/*****
**
** bitebits.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $new_dbits_wo
** $new_dbits_nwo
** $new_dbits2
** $do_syms
** $encode_symbol
** I_DIV_JW
** $update_model
** $bit_plus_follow
** $new_output_bit
**
*****/

.global $new_dbits_wo
.global $new_dbits_nwo

.global $new_dbits2

.global $output_bit
.global $do_syms
.global $update_model
.global $bit_plus_follow

.global $T_BYTES
.global $byte_stream
.global $STOP

.global $bit_index

.global $ztl                ;signed short pointer sh *(xba + $ztl)
.global $THRESH             ;signed int sw *(xba + $THRESH)
.global $stats_flag         ;unsigned char pointer *(xba + $stats_flag)
.global $char_to_index      ;unsigned char pointer &*(xba + $char_to_index)
.global $stats_val          ;signed short pointer *(xba + $stats_val)
.global $TMASK              ;unsigned short uh *(xba + $TMASK)
.global $BITE               ;signed int sw *(xba + $BITE)
.global $SHFT_DN            ;signed int sw *(xba + $SHFT_DN)

.global $update_model
.global $encode_symbol

.global $new_output_bit

.global $list
.global $list_index

.global $pruned_children

.global $getaway_address
.global $quick_getaway

.global $index_to_char
.global $No_of_symbols
.global $bits_to_follow

.global $high
.global $freq
.global $low
.global $cum_freq

.global $bit_plus_follow

```

```

.global $sym_array
.global $sym_index

.global $stophere

tempd1      .set    d0
stats_flag  .set    d1
stats_val   .set    d2
tempd2      .set    d3
tflg       .set    d4
tmask      .set    d5
tempd3      .set    d6
t          .set    d7
sym         .set    a12

maxsyms     .set    240

.align 512

/*****
*
* Function : $new_dbits_wo
* Args    : m ,c1,c3,s
* Passed in : (d1,d2,d3,d4)
*
* Description:
*   m - the current index of this coefficient
*
*   c1 - the relative index of the first child
*
*   c3 - the relative index of the third child
*
*   s - the subblock that we are in
*
* $new_dbits_wo will be called to scan a coefficient that is
* on the second or third tier of the tree. The coefficient
* has already been checked previously for being significant
* on a previous bit-plane, now it is check against the
* current bit-plane threshold for significance, or pass
* down.
*
* Return Values:
*   None
*
*****/

$new_dbits_wo:

    d0 = &(sp --- 28)
    *(sp + 16) =w iprs

    *(sp + 12) =w a4
    || *(sp + 8) =w d6

    *(sp + 4) =w a12
    || *(sp + 0) =w d7

    *(sp + 24) = d4                ;save s onto stack

    d5 = d1

    a1 =uw *(xba + $stats_val)      ;*(a1) = stats_val[s*256]
    a2 =uw *(xba + $stats_flag)     ;*(a3 + [x0]) = ztl[]
    a3 =uw *(xba + $ztl)

    d6 = d4 << 9                    ; 256 elements each 2 bytes long times s
    (<< 9 = * 512)
    a1 = a1 + d6

```

# NAWCWD TP 8442

```

        d6 = d4 << 8                ; 256 elements each 1 byte long times s
(<< 8 = * 256)
        a2 = a2 + d6

        d6 = d4 << 7                ;ztl[s*64] (where ztl[] is a short)
        a3 = a3 + d6

        d5 = d5 + (d4<<8)           ;m = m + s*256

        *(sp + 20) =uh d5

        a0 = a2                      ;*(a0)          = stats_flag[s*256]

        x0 = d1                      ;x0 = m
        x1 = d3                      ;*(a2 + [x1]) = stats_flag[c3]
        x2 = d3 + 1                  ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

        a2 = a2 + d2                 ;*(a2)          = stats_flag[s*256 + c1]
                                      ;*(a2 + [1])      = stats_flag[c2 = c1 + 1]

        ;*(a1 + [x0]) = stats_val[s*256 + m]
        ;*(a0 + [x0]) = stats_flag[s*256 + m]
        ;*(a2)          = stats_flag[s*256 + c1]
        ;*(a2 + [1])    = stats_flag[s*256 + c2] (c2 = c1 + 1)
        ;*(a2 + [x1])   = stats_flag[s*256 + c3]
        ;*(a2 + [x2])   = stats_flag[s*256 + c4] (c4 = c3 + 1)

        tmask =uh *(xba + $TMASK)

        tflg =sh *(a3 + [x0])         ;fetch ztl[m + s*64]
        a15 = tflg&tmask
        tflg = 1 || tflg =[ne] a15    ;calculate tflg

        tflg = tflg - 0
        br =[z] no_change
        stats_val =sh *(a1 + [x0])    ;fetch stats_val[m]
        tempd3 = |stats_val|          ;take abs(stats_val[m])

        tempd1 = tflg << 1
        a4 = tflg

        call = mod_flags
        d4 = *(sp + 24)                ;reload s from stack into d4
        tempd2 =ub *(a2)

        tflg = a4

no_change:
        t = tempd3 & tmask             ;form TMASK&abs(stats_val[m])
        tempd1 =sw *(xba + $SHFT_DN)  ;fetch SHFT_DN
        tempd1 = -tempd1
        t = t >>u -tempd1             ;form (TMASK&abs(stats_val[m]))>>SHFT_DN

;compute sym

        tempd1 =sw *(xba + $BITE)
        tempd1 = %tempd1
        t = t - 0
        tempd1 =[eq] a15
        stats_val = stats_val - 0
        tempd1 =[le] a15

finish_up:
        tempd2 = (~tflg)&1
        tempd2 = tempd2 + t
        tempd2 = tempd2 + tempd1

        x1 = tempd2

```



# NAWCWD TP 8442

```

t = t - 0
br =[le] L68
nop
nop

; if (t>0)
; stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)/(1<<(BITE-1)) + THRESH/(2<<(BITE-1)))));
; /* or the equivalent from Chuck's code */
; stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)>>(BITE-1)) + (THRESH>>BITE));

tempd1 =sw *(xba + $BITE)
tempd2 =uh *(xba + $THRESH)

tflg = 1 - tempd1

tmask =u tempd2 * t
tempd1 = - tempd1

tempd3 = tempd3 - (tmask >>u -tflg)
tempd3 = tempd3 - (tempd2 >>u -tempd1)
*(a1 + [x0]) =h tempd3

L68:
tempd1 = 0
t = t - 0
tempd1 =[nz] 4 ; calculate (t!=0)<<2
*(a0 + [x0]) =ub tempd1

a15 = tempd1 - 0
br =[eq] nosig
x0 =uh *(pba + $list_index)
a15 = x0 - 254
br =[ge] nosig
tempd1 =uh *(sp + 20)
a0 =uw *(pba + $list)
tempd2 = x0 + 1
*(a0 + [x0]) =uh tempd1
*(pba + $list_index) =uh tempd2

nosig:
a0 = &*(pba + $sym_array)
x0 =uh *(pba + $sym_index)

nop

tempd2 = x0 + 1
*(pba + $sym_index) =uh tempd2

; sym_array[sym_index++] = sym;

*(a0 + [x0]) =ub x1

; if (sym_index>240) do_syms();

tempd1 = tempd2 - maxsyms

call =[gt] $do_syms
nop
nop

done_more:
a4 = *(sp + 12)
br = *(sp + 16)

d6 =sw *(sp + 8)
|| d7 =sw *(sp + 0)
d0 = &*(sp += 28)
; branch occurs here

```

```

/*****
*
* Function : $new_dbits_nwo
* Args : m ,c1,c3,s
* Passed in : (d1,d2,d3,d4)
*
* Description:
*   m - the current index of this coefficient
*
*   c1 - the relative index of the first child
*
*   c3 - the relative index of the third child
*
*   s - the subblock that we are in
*
* $new_dbits_nwo will be called when the current coefficient
*   and all of its children have been deemed insignificant,
*   from the pass down flag.
*
* Return Values:
*   None
*
*****/

$new_dbits_nwo:

    a2 =uw *(xba + $stats_flag)
    d5 = d4 << 8
    a2 = a2 + d5

    a0 = a2                                ;*(a0 + [x0]) = stats_flag[s*256 + m]

    x0 = d1                                ;x0 = m
    x1 = d3                                ;*(a2 + [x1]) = stats_flag[s*256 + c3]
    x2 = d3 + 1                            ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

    a2 = a2 + d2                            ;*(a2) = stats_flag[s*256 + c1]
                                           ;*(a2 + [1]) = stats_flag[c2 = c1 + 1]

    tempd2 =ub *(a0 + [x0])

    tempd2 = tempd2 & 252
    *(a0 + [x0]) =ub tempd2

    tempd1 = 2
    tempd2 =ub *(a2)

mod_flags:

    tempd2 = tempd2 | tempd1
    *(a2) =ub tempd2

    tempd2 =ub *(a2 + [1])
    tempd2 = tempd2 | tempd1
    *(a2 + [1]) =ub tempd2

    tempd2 =ub *(a2 + [x1])
    tempd2 = tempd2 | tempd1
    *(a2 + [x1]) =ub tempd2

    tempd2 =ub *(a2 + [x2])

    tempd2 = tempd2 | tempd1
    *(a2 + [x2]) =ub tempd2

    a2 = &*(pba + $pruned_children)
    x2 = d4                                ;s is in d4

```

# NAWCWD TP 8442

```

nop
tempd1 =ub *(a2 + x2)

br = iprs

tempd1 = tempd1 + 1
*(a2 + x2) =ub tempd1

.align 512

/*****
*
* Function : $new_dbits2
* Args : m,s
* Passed in :d1,d2
*
* Description:
* $new_dbits2 will be called to scan the fourth tier in the
* zero tree.
*
* Return Values:
* None
*
*****/

$new_dbits2:

    d0 = &*(sp --- 24)
    *(sp + 16) =w iprs

    *(sp + 8) =w a4
    *(sp + 4) =w d6

    *(sp + 0) =w d7

    a0 =uw *(xba + $stats_flag)
    a1 =uw *(xba + $stats_val) ;*(a1) = stats_val[s*256]

    d5 = d1

    d3 = d2 << 9
    a1 = a1 + d3

    d3 = d2 << 8
    a0 = a0 + d3

    d5 = d5 + (d2 << 8)

    *(sp + 20) =uh d5 ;index = m + s*256

    x0 = d1 ;x0 = m

    ;*(a0 + [x0]) = stats_flag[s*768 + m]
    ;*(a1 + [x0]) = stats_val[s*768 + m]

    tmask =uh *(xba + $TMASK)

    stats_val =sh *(a1 + [x0]) ;fetch stats_val[m]
    tempd3 = |stats_val| ;take abs(stats_val[m])
    t = tempd3 & tmask ;form TMASK&abs(stats_val[m])

    tempd1 =sw *(xba + $SHFT_DN) ;fetch SHFT_DN
    tempd1 = -tempd1
    t = t >>u -tempd1 ;form
    (TMASK&abs(stats_val[m]))>>SHFT_DN

```

```

;compute symbol

    tempd2 =sw *(xba + $BITE)
    tempd2 = %tempd2

    stats_val = stats_val - 0
    tempd2 =[le] a15

    t = t - 0
    tempd2 =g[eq] a15

    tempd2 =[ne] tempd2 + 1 ;form (t!=0) and add to (1<<BITE)-1

    t = t - 0
    br =[le] L69

    x1 = tempd2 + t ;sym = t + (((t!=0)&&(stats_val[m]>0))
? ((1<<BITE)-1):0) + (t!=0)
    a4 = &*(xba + $char_to_index)

;if (t>0)
; stats_val[m] = (abs(stats_val[m])-((t*THRESH)>>(BITE-1))+(THRESH>>BITE));

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

    tflg = 1 - tempd1

    tmask =u tempd2 * t
    tempd1 = - tempd1

    tempd3 = tempd3 - (tmask >>u -tflg)
    tempd3 = tempd3 - (tempd2 >>u -tempd1)
    *(a1 + [x0]) =h tempd3

L69:
    tempd1 = 0
    t = t - 0
    tempd1 =[nz] 4 ;calculate (t!=0)<<2
    *(a0 + [x0]) =ub tempd1

    a15 = tempd1 - 0
    br =[eq] nosig2
    x0 =uh *(pba + $list_index)
    a15 = x0 - 254
    br =[ge] nosig2
    tempd1 =uh *(sp + 20)
    a0 =uw *(pba + $list)
    tempd2 = x0 + 1
    *(a0 + [x0]) =uh tempd1
    *(pba + $list_index) =uh tempd2

nosig2:
    a0 = &*(pba + $sym_array)
    x0 =uh *(pba + $sym_index)

    tempd2 = x0 + 1
    *(pba + $sym_index) =uh tempd2

;sym_array[sym_index++] = sym;

    *(a0 + [x0]) =ub x1

; if (sym_index>240) do_syms();

    tempd1 = tempd2 - maxsyms

    call =[gt] $do_syms

```

# NAWCWD TP 8442

```

nop
nop

a4 =sw *(sp + 8)

br = *(sp + 16)

d6 =sw *(sp + 4)
|| d7 =sw *(sp + 0)
d0 = &*(sp += 24)

.align 512

/*****
*
* Function : do_syms
* Args : none
*
* Description:
* do_syms will be called to empty the symbol cache.
*
* Return Values:
* None
*
*****/

$do_syms:
    d0 = &*(sp --= 20)
    *(sp + 16) =w iprs

    *(sp + 12) =w a4
||    *(sp + 8) =w d6

    *(sp + 4) =w a12
||    *(sp + 0) =w d7

    *(xba + $stophere) =uw a15

    a12 = &*(xba + $char_to_index)
    a4 = &*(xba + $sym_array)

    d6 =uh *(xba + $sym_index)
    x8 =ub *a4++
    nop

more_syms:
    d7 =ub *(a12 + x8)

    call = $encode_symbol
    nop
    d1 = d7

    call = $update_model
    nop
    d1 = d7

    d1 =sw *(xba + $STOP)
    d1 = d1 - 1
    br =[eq] .get_out2
    nop

    d6 = d6 - 1
    br =[gt] more_syms
    x8 =ub *a4++
    nop

```

get\_out:

```

    a4 =w *(sp + 12)
||    d6 =w *(sp + 8)

    a12 =w *(sp + 4)
||    d7 =w *(sp + 0)

    br = *(sp + 16)
    *(pba + $sym_index) =uh a15
    d0 = &*(sp += 20)

```

get\_out2:

```

    a4 =w *(sp + 12)
||    d6 =w *(sp + 8)

    a12 =w *(sp + 4)
||    d7 =w *(sp + 0)

    d0 = *(sp + 16)
    d2 = *(pba + $getaway_address)
    d1 =ub *(pba + $quick_getaway)
    d1 = d1 - 1
    d0 =[eq] d2

    br = d0
    *(pba + $sym_index) =uh a15
    d0 = &*(sp += 20)

```

```

/*****
*
*   Function : encode_symbol
*   Args      : none
*
*   Description:
*       encode_symbol will be called to perform the arithmetic
*       encoding of a symbol. This routine was lifted from a C
*       compiled program and included here, for cache coherency.
*
*   Return Values:
*       None
*
*****/

```

\$encode\_symbol:

```

    x0 = d1
    a0 = &*(xba + $cum_freq)
    d3 =sw *(xba + $low)
    d2 =sw *(xba + $high)
    d1 = x0 << 1

    d4 = d2 - d3
||    d2 =g a0

    a1 = d1 + d2
||    d0 = &*(sp --- 4)

    d4 = d4 + 1
||    *(sp) =w iprs

    d1 =uh1 d4
||    d5 =sh *(a1 - 2)

    d1 =u d5 * d1
||    d2 =uh1 d5
    d2 =u d2 * d4

```

```

    d1 =u d5 * d4
    d2 = d2 + d1
    call = I_DIV_JW

    d1 = d1 + (d2 << 16)
    x1 =sh *a0
    d2 = x1

    d0 =uh1 d4
    d1 =sh *(a0 + [x0])

    d0 =u d0 * d1
    d2 =uh1 d1
    d2 =u d4 * d2

    d1 =u d4 * d1
    d2 = d0 + d2
    d1 = d1 + (d2 << 16)

    d4 = d5 + d3
    d2 = x1
    call = I_DIV_JW
    d4 = d4 - 1
    *(xba + $high) =w d4

    d2 = d5 + d3
    d1 = d4 - (1 \\ 8)
    br =[ge] L25
    *(xba + $low) =w d2
    d1 =[ge] d2 - (1 \\ 8)

L19:
    call = $bit_plus_follow
    nop
    d1 = 0

    br = L23
    d1 =sw *(xba + $high)
    d1 = d1 << 1

L20:
    d1 =sh *(xba + $bits_to_follow)

    d2 = d1 + 1
    d3 =sw *(xba + $low)
    *(xba + $bits_to_follow) =h d2

    d2 = d3 - (1 \\ 7)
    d1 =sw *(xba + $high)
    br = L22

    d1 = d1 - (1 \\ 7)
    *(xba + $low) =w d2
    *(xba + $high) =w d1

L21:
    call = $bit_plus_follow
    nop
    d1 = 1

    d1 =sw *(xba + $low)

    d1 = d1 - (1 \\ 8)
    d2 =sw *(xba + $high)

    d1 = d2 - (1 \\ 8)
    *(xba + $low) =w d1
    *(xba + $high) =w d1

```

```

L22:      d1 = d1 << 1

L23:      d4 = d1 + 1
||      d1 =sw *(xba + $low)
        d2 = d1 << 1
        d1 = d4 - (1 \\ 8)
        br =[lt] L19
        *(xba + $high) =w d4
        *(xba + $low) =w d2

L24:      d1 = d2 - (1 \\ 8)

L25:      br =[ge] L21
        nop
        d1 =[lt] d2 - (1 \\ 7)

        br =[lt] L30
        nop
        d1 =[ge] d4 - 384

        br =[lt] L20
        br =[ge] L30
        nop

        nop

L30:      br = *(sp)
        nop
        d0 = &*(sp += 4)

```

```

/*****
*
*   Function : I_DIV_JW
*   Args      : none
*
*   Description:
*       I_DIV_JW will be called to perform an Integer Divide.
*
*   Return Values:
*       None
*
*****/

```

```

;*****
;* I_DIV.ASM  v1.10  - Integer Divide
;* Copyright (c) 1993-1995 Texas Instruments Incorporated
;*****

```

```

; +-----+
; |       i_div.asm = PP assembly program that is used to return a 32-bit |
; |               signed integer quotient from 32-bit signed integer   |
; |               division when called by a C program.                  |
; |                                                                     |
; +-----+

```

```

.global    I_DIV_JW

```



```

; +-----+
; | 32-bit Signed Integer Word Divide Subroutine : |
; |   o Input 32-bit signed integer Operand 1 is in d1 (numerator). |
; |   o Input 32-bit signed integer Operand 2 is in d2 (divisor). |
; |   o Output 32-bit signed integer is in d5 (Answer = quotient). |
; |   o Output 32-bit signed remainder is discarded. |
; |   o 0 input divisor produces 0x80000000 output with overflow set. |
; |   o Quotient = 0x80000000 sets overflow. |
; |   o Number of Stack Words used = 3. |
; |   o MF register is saved. |
; | |
; |   o NOTE: Loop Counter 2 Registers are used but NOT restored ! |
; +-----+

```

```

; +-----+
; | 32 bit / 32 bit ==> 32 bit signed quotient |
; | Signed PP Integer Division |
; | Numerator / Denominator = Quotient + Remainder (discarded) |
; | Divide by 0 produces 80000000 and sets sr(V) |
; | Divide Overflow is not possible if Divisor is non-zero, |
; | except 80000000/ffffffff = 80000000 will set sr(V). |
; | MF register is preserved. |
; +-----+

```

```

arg1: .set d1 ; input argument 1 = Numerator (32 low bits)
arg2: .set d2 ; input argument 2 = Divisor (32 bits)
ans: .set d5 ; answer = 32 bit signed quotient
Div: .set d3 ; Input Divisor
Num: .set d4 ; Input high Numerator = 0
Tmp: .set d5 ; ALU output for each DIVI

```

I\_DIV\_JW: ; Signed Word Integer Divide: Ans = Op1 / Op2

```

Div = 0 - | arg2 | ; negate | divisor |
|| *(sp-=|3|) = Div ; || push Div
br = [z] Div_By_0 ; Divide By 0 ?
Num = 0 ; high numerator = 0
|| *(sp+|1|) = mf ; || push mf
|| *(sp+|2|) = Num ; || push Num
mf = | arg1 | ; input lo | numerator |

lrse2 = 29 ; loop count - 1
Tmp = divi(Div, Num=Num) ; 1-st divide iterate
Tmp = divi(Div, Num=Tmp [n] Num) ; 2-nd divide iterate
LoopSW: Tmp = divi(Div, Num=Tmp [n] Num) ; divide iterate 3-32

ans = mf ; | ans | = mf
|| Div = *sp++ ; || pop Div
Num = arg1 ^ arg2 ; quotient sign
|| br = iprs ; || return
ans = [n] -ans ; quotient is negative,
|| mf = *sp++ ; || pop mf
Num = *sp++ ; pop Num

```

```

Div_By_0: ; Divide By 0 \ Optional Error
Div_Ovfl: ; Divide Overflow / Return Code
br = iprs ; return
|| Div = *sp++ ; || pop Div
mf = *sp++ ; pop mf
ans = 0 - 1<<31 ; returns 0x80000000, sets sr(V)
|| Num = *sp++ ; || pop Num ...[END]

```

```

/*****
*
*   Function : update_model
*   Args      : none
*
*   Description:
*       update_model will be called to update the arithmetic
*       model's parameters. This routine was lifted from a C
*       compiled program and included here, for cache coherency.
*
*   Return Values:
*       None
*
*****/

$update_model:
    a0 = &(xba + $cum_freq)
    nop

    a1 = d1
||    d1 =sh  *a0
    d1 = d1 - 75
    br =[ne] L6
    nop
    a2 =g [ne.ncvz] a1

    d2 =sw  *(xba + $No_of_symbols)
    d1 = d2 - 0
    br =[lt] L6
    nop
    a2 =g [lt.ncvz] a1

    d1 =g  a0
    le1 = L5 - 8

    d5 = d2 << 1
||    d3 = &(xba + $freq)
    a0 = d5 + d1

    d4 = d2 << 1
||    lrs1 = d2
    a8 = d4 + d3
    d2 = 0

L4:
    d1 =sh  *a8
    d1 = d1 + 1
    d3 = d1 >>u 31
    d1 = d1 + d3
    d1 = d1 >>s 1
    d1 =sh0 d1
    *a8 =h  d1
    *a0-- =h d2
    d1 =sh  *a8--
    d2 = d2 + d1

L5:
    a2 =g  a1

L6:
    d3 = &(xba + $freq)
    d1 = a2 << 1
    d5 = a2 << 1
    d4 = d3 + d1
    a0 = d5 + d3
    a8 = d4 - 2
    nop

```

# NAWCWD TP 8442

```

    d4 =sh  *a8
||    d3 =sh  *a0
    d3 = d3 - d4
    br =[ne] L9
    d2 =g [ne.ncvz] a2
    d3 =[ne] d2 - a1

L7:
    d1 = d1 - 2
||    d3 =sh  *---a8

    a2 = a2 - 1
||    d4 =sh  *---a0
    d3 = d4 - d3
    br =[eq] L7
    nop
    nop

L8:
    d2 =g  a2
    d3 = d2 - a1

L9:
    br =[ge] L11
    nop
    nop

    x0 = &*(xba + $index_to_char)
    nop
    a8 =ub  *(a2 + x0)
    x8 = &*(xba + $char_to_index)
    a9 =ub  *(a1 + x0)
    *(a2 + x0) =b  a9
    *(a1 + x0) =b  a8
    *(a8 + x8) =b  a1
    *(a9 + x8) =b  a2

L11:
    d3 =sh  *a0
    d3 = d3 + 1

||    d3 = a2 - 0
    *a0 =h  d3
    br =[le] L15
    br =[le] L16
    nop

    le2 = L14 - 8
    d2 = a2 - 1
    d3 = &*(xba + $cum_freq)
    lrs2 = d2
    a0 = d1 + d3
    nop

L13:
    d1 =sh  *--a0
    d1 = d1 + 1
    *a0 =h  d1

L14:
    br = L16
    nop

L15:
    nop

L16:
    br = iprs
    nop
    nop

```

```

/*****
*
*   Function : bit_plus_follow
*   Args      : none
*
*   Description:
*       bit_plus_follow will be called to output several bits that
*       have been encoded by the arithmetic encoder.
*
*   Return Values:
*       None
*
*****/

$bit_plus_follow:
    d0 = &(sp --= 8)
    *(sp + 4) =w iprs
    call = $new_output_bit
    nop

    d6 = d1
||    *(sp + 0) =w d6

    d1 =sh *(xba + $bits_to_follow)
    d1 = d1 - 0
    br =[le] L36
    nop
    d1 =[gt] d6 - 0

    d6 = 1 || d6 =[ne] a15
L34:    call = $new_output_bit
    nop
    d1 = d6

    d1 =sh *(xba + $bits_to_follow)
    d1 = d1 - 1
    d1 =sh0 d1

    d1 = d1 - 0
||    *(xba + $bits_to_follow) =h d1
    br =[gt] L34
    nop
    nop

L36:    br = *(sp + 4)
    nop
    d6 =sw *(sp + 0)
||    d0 = &(sp += 8)

```

```

/*****
*
*  Function : new_output_bit
*  Args      : d1 - the bit to append
*
*  Description:
*      new_output_bit will be called to append a single bit to
*      the bitstream array.
*
*  Return Values:
*      None
*
*****/

```

```

$new_output_bit:

```

```

    d2 =sw  *(xba + $STOP)
    d2 = d2 - 1
    br =[eq] iprs
    d2 =uh  *(xba + $bit_index)
    d4 = d2 >>u 3
||    d3 =uw  *(xba + $byte_stream)

    a0 = d4 + d3
    d0 =uh  *(xba + $T_BYTES)

    d4 = (d2&7)
||    d3 =ub  *a0
    d3 = d3 | (d1 << d4)
    *a0 =b  d3
||    d5 = d2 + 1

    *(xba + $bit_index) =h  d5

    d5 = d5 - (d0 << 3)

    br =g iprs
    d1 = 1 || d1 =[lt] a15      ;calculate new STOP value
    *(xba + $STOP) =w  d1

```

# NAWCWD TP 8442

```
mpcl -s -g -c -i\pcic80\include mp.c  
mvplnk -x mp.obj num2.obj pp0.out pcic80.cmd -o mp.out -m mp.map
```

## NAWCWD TP 8442

```
erase *.o
ppcl -s -k main.c
ppcl -s -k j_enc.c
ppcl -s -k modlp.c
ppasm coded.s
ppasm bitebits.s
ppasm subpass.s
ppasm hvenc.s
ppasm mean2.s
ppasm ztr.s
mvplnk -x main.o hvenc.o mean2.o j_enc.o coded.o ztr.o bitebits.o subpass.o modlp.o
pcic80a.cmd -t runpp0 -o pp0.out -m pp0.map
```

```

/*****
**
** codenew.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains code_dlist2 subroutine.
**
*****/

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

#define XSIZE 512
#define YSIZE 256
#define NSCALES 5
#define AX 16 /* = XSIZE/BS */
#define AY 8 /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */
#define maxsyms 240

extern unsigned char emubrk;
extern unsigned char *stats_flag;
extern int STOP;
extern unsigned short syms_to_do;
extern unsigned short *save_array;
extern unsigned char read_more;
extern unsigned short left_off;
extern int FIRST_DEC;
extern unsigned char RB_DEC;
extern unsigned char PASS_DEC;
extern unsigned short THRESH;
extern unsigned short *list;
extern unsigned short list_index;
extern unsigned char index_to_char[Max_No_of_symbols+1];
extern short *stats_val;
extern unsigned short symbols_read;
extern int BITE;
extern unsigned char PASS_ENC;
extern unsigned short TMASK;
extern int SHFT_DN,RB_ENC;
extern unsigned char sym_array[256];
extern unsigned short sym_index;
extern unsigned char pruned_children[8]; /* used to keep count of the number of pruned */
/* children within a zero tree */
extern unsigned char total_links; /* keeps count of pruned trees */
extern unsigned char link_list[6]; /* list with unpruned trees in it */

/*****
*
* Function : code_dlist2
* Args : none
*
* Description:
* code_dlist2 scans six zero trees in order for significant
* coefficients, it also prunes off zero trees that have no
* significant coefficients below the current level.
*
* Return Values:
* None
*
*****/

void code_dlist2()
{
int i,p,s;
unsigned short t;

```



```

register unsigned int m,c;

/* Perform Dominant pass */

for (s=0;s<6;s++) pruned_children[s] = 0;

for (s=0;s<6;s++) comp_dbits(0,1,2,3,s);

m = 0;

for (s=0;s<6;s++)
{
    if (pruned_children[s] < 1)
    {
        link_list[m++] = s;
        pruned_children[s] = 0;
    }
    else
    {
        pruned_children[s] = 255;
        for (i=(s*256)+1;i<((s*256)+4);i++)
            stats_flag[i] = stats_flag[i] & 252;
    }
}

total_links = m;

if ((STOP==0) && (total_links!=0))
    for(m=1;m<4;m++)
    {
        c = 4*m;
        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
            if ((stats_flag[m + s*256]&2)==2)
                new_dbits_nwo(m,c,2,s);
            else if ((stats_flag[m + s*256]&6)==0)
                new_dbits_wo(m,c,2,s);
        }
    }

m = 0;

for (s=0;s<6;s++)
{
    if (pruned_children[s] < 3 )
    {
        link_list[m++] = s;
        pruned_children[s] = 0;
    }
    else
    {
        if (pruned_children[s] != 255)
            for (i=(s*256+4);i<(s*256+16);i++)
                stats_flag[i] = stats_flag[i] & 252;
        pruned_children[s] = 255;
    }
}

total_links = m;

if ((STOP==0) && (total_links!=0))
    for(m=4;m<16;m++)
    {
        c = 8*(m/2) + 2*(m%2);
        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
            if ((stats_flag[m + s*256]&2)==2)

```

```

        new_dbits_nwo(m,c,4,s);
    else if ((stats_flag[m + s*256]&6)==0)
        new_dbits_wo(m,c,4,s);
    }
}

m = 0;

for (s=0;s<6;s++)
{
    if (pruned_children[s] < 12 )
    {
        link_list[m++] = s;
        pruned_children[s] = 0;
    }
    else
    {
        if (pruned_children[s] != 255)
            for (i=(s*256+16);i<(s*256+64);i++)
                stats_flag[i] = stats_flag[i] & 252;
        pruned_children[s] = 255;
    }
}

total_links = m;

if ((STOP==0) && (total_links!=0))
    for(m=16;m<64;m++)
    {
        c = 16*(m/4) + 2*(m%4);
        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
            if ((stats_flag[m + s*256]&2)==2)
                new_dbits_nwo(m,c,8,s);
            else if ((stats_flag[m + s*256]&6)==0)
                new_dbits_wo(m,c,8,s);
        }
    }

m = 0;

for (s=0;s<6;s++)
{
    if (pruned_children[s] < 48 )
    {
        link_list[m++] = s;
        pruned_children[s] = 0;
    }
    else
    {
        if (pruned_children[s] != 255)
            for (i=(s*256+64);i<(s*256+256);i++)
                stats_flag[i] = stats_flag[i] & 252;
        pruned_children[s] = 255;
    }
}

total_links = m;

if ((STOP==0) && (total_links!=0))
    for(m=64;m<256;m++)
    {
        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
            if ((stats_flag[m+s*256]&2)==2)
                stats_flag[m+s*256] = stats_flag[m+s*256] & 252;
            else if ((stats_flag[m+s*256]&6)==0)

```

```

        new_dbits2(m,s);
    }
}

if (sym_index>0)
    do_syms();

sym_index = 0;

start_model(2);

if (STOP==0)
    PASS_ENC += BITE;

t = THRESH>>BITE;

THRESH = THRESH>>BITE;

if (PASS_ENC == BITE)
{
    BITE = RB_ENC;
    TMASK = THRESH;
    for(m=1;m<BITE;m++)
        TMASK = TMASK | (THRESH>>m);
}
else
{
    BITE = 1;
    TMASK = THRESH;
}

SHFT_DN -= BITE;

/* Subordinate Pass */

subpass(t);

if (sym_index>0)
    do_syms();

sym_index = 0;

start_model(1<<(BITE+1));

}

```

```

/*****
**
** hvenc.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $subdec_vert
** $subdec_horiz
**
*****/

        .global      $subdec_vert
        .global      $subdec_horiz

        .align 2048

/*****
*
* Function : $subdec_vert
* Args      : outerloop,innerloop
* Passed in : (d1      ,d2)
*
* Description:
*   outerloop - the width of the coefficient patch
*
*   innerloop - the height of the coefficient patch
*
*   $subdec_vert will be called to perform the wavelet
*   decomposition in the vertical direction.
*
* Return Values:
*   None
*
*****/

$subdec_vert:
    lctl = 0x0 ;reset looping capability

    lr0 = d1 - 1      ;outerloop
    lr1 = d2 - 3      ;innerloop
    a4 = d1

    d7 = 0            ; k = 0

    le1 = InnerLoopEnd ;
    ls1 = InnerLoop    ;

    le0 = OuterLoopEnd
    ls0 = OuterLoop
    nop
    lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

    d3 = a4
    d4 = d3 + d3
    x1 = d4
    d4 = d4 + d3
    x2 = d4

OuterLoop:
;    img[index][k] = (img[index][k]>>1) - ((img[0][k]+img[2*index][k])>>2);

    nop
    x0 = d7
    nop
    a0 =h &*(dba + [x0])
    nop
    x0 = a4

```

```

        d2 =sh *(a0 + [x1])
        d3 =sh *(a0)

        d4 =sh *(a0 + [x0])
||      d2 = d2 + d3

        d4 = (d4 >>s 1)
        d4 = d4 - (d2 >>s 2)
        *(a0 + [x0]) =h d4

;      img[0][k] += img[index][k];

        d3 = d3 + d4
        *(a0++=[x0]) =h d3
        nop

InnerLoop:

;      img[2*1+index][k] = (img[2*1+index][k]>>1) -
((img[2*1][k]+img[2*1+2*index][k])>>2);

        d3 =sh *(a0 + [x0])
        d4 =sh *(a0 + [x2])

||      d2 =sh *(a0 + [x1])
        d5 = d3 + d4

        d2 = (d2 >>s 1)
        d2 = d2 - (d5 >>s 2)

        *(a0 + [x1]) =h d2

;      img[2*1][k] += ((img[2*1+index][k]+img[2*1-index][k]+2)>>1);

        d4 =sh *(a0++=[x0])
        d5 = d2 + d4
        d3 = d3 + (d5 >>s 1)

        *(a0++=[x0]) =h d3

InnerLoopEnd:
        nop

;      img[YSIZE-index][k] = (img[YSIZE-index][k]-img[YSIZE-2*index][k])>>1;

        d2 =sh *(a0 + [x1])
        d3 =sh *(a0 + [x0])
        d2 = d2 - d3
        d2 = (d2 >>s 1)
        *(a0 + [x1]) =h d2

;      img[YSIZE-2*index][k] += ((img[YSIZE-index][k]+img[YSIZE-3*index][k]+2)>>1);

        d4 =sh *(a0)
        d5 = d2 + d4
        d3 = d3 + (d5 >>s 1)

        *(a0 + [x0]) =h d3

OuterLoopEnd:
        d7 = d7 + 1

        br = iprs
        nop
        nop

```

```

/*****

```

# NAWCWD TP 8442

```

*
* Function : $subdec_horiz
* Args    : outerloop,innerloop
* Passed in : (d1      ,d2)
*
* Description:
*   outerloop - the width of the coefficient patch
*   innerloop - the height of the coefficient patch
*
*   $subdec_horiz will be called to perform the wavelet
*   decomposition in the horizontal direction.
*
* Return Values:
*   None
*
*****/

$subdec_horiz:

    lctl = 0x0          ;reset looping capability

    lr0 = d1 - 1
    lr1 = d2 - 3

    a4 = d2             ; innerloop

    d7 = 0              ; k = 0

    le1 = InnerLoopEnd2
    ls1 = InnerLoop2

    le0 = OuterLoopEnd2
    ls0 = OuterLoop2

    nop

    lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

    nop
    nop

OuterLoop2:
;   img[k][index] -= ((img[k][0]+img[k][2*index])>>1);
;           a0           a1           a2

    d3 = a4
    d3 = d3 * d7
    d3 = d3 << 1
    x0 = d3
    nop
    a0 =h &*(dba + [x0])
    nop

    d3 =sh *(a0)
    d2 =sh *(a0 + [2])

    d4 =sh *(a0 + [1])
    d2 = d2 + d3

    d4 = d4 - (d2 >>s 1)
    *(a0 + [1]) =h d4

;   img[k][0] = (img[k][0]<<1) + img[k][index];

    d4 = d4 + (d3 << 1)
    *(a0++=[1]) =h d4

    nop

```

# NAWCWD TP 8442

InnerLoop2:

```
;      img[k][2*1+index] -= ((img[k][2*1]+img[k][2*1+2*index])>>1);

      d3 =sh *(a0 + [1])
      d4 =sh *(a0 + [3])

      d2 =sh *(a0 + [2])
||     d5 = d3 + d4

      d2 = d2 - (d5 >>s 1)

      *(a0 + [2]) =h d2

;      img[k][2*1] = (img[k][2*1]<<1) + ((img[k][2*1+index]+img[k][2*1-index])>>1);

      d4 =sh *(a0++=[1])
      d5 = d2 + d4
      d3 = d3 << 1
      d3 = d3 + (d5 >>s 1)

      *(a0++=[1]) =h d3
```

InnerLoopEnd2:

```
      nop

;      img[k][XSIZE-index] -= img[k][XSIZE-2*index];

      d2 =sh *(a0 + [2])
      d3 =sh *(a0 + [1])
      d2 = d2 - d3
      *(a0 + [2]) =h d2

;      img[k][XSIZE-2*index] = (img[k][XSIZE-2*index]<<1) + ((img[k][XSIZE-
index]+img[k][XSIZE-3*index])>>1);

      d4 =sh *(a0)
      d5 = d2 + d4
      d3 = d3 << 1
      d3 = d3 + (d5 >>s 1)

      *(a0 + [1]) =h d3
```

OuterLoopEnd2:

```
      d7 = d7 + 1

      br = iprs
      nop
      nop
```

```

/*****
**
** j_enc.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** PP Program that orchestrates the generation of the bitstream. Processes
** 6 Zero Trees per partition.
**
*****/

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

#define ASSEM
#define ASSEM3
#define ASSEM2

#define XSIZE 512
#define YSIZE 256
#define NSCALES 5
#define AX 16 /* = XSIZE/BS */
#define AY 8 /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

#define maxsyms 240

unsigned char sym_array[256];
unsigned short sym_index;

int stophere;

unsigned char PASS_ENC;
unsigned short TMASK;

int SHFT_DN, RB_ENC;

unsigned char buffer; /* Bits buffered for output */
unsigned char bits_to_go; /* # bits free in buffer */

unsigned char pruned_children[8]; /* used to keep count of the number of pruned */
/* children within a zero tree */
unsigned char total_links; /* keeps count of pruned trees */
unsigned char link_list[6]; /* list with unpruned trees in it */

unsigned short list_index;

unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

/* CURRENT STATE OF ENCODING */

int low, high; /* Ends of current code region */
short bits_to_follow; /* Number of opposite bits to output
after the next bits */

extern shared int local_maxval[4];
extern shared int global_maxval;

extern short *img;
extern short *coeff_block;
extern short *stats_val;
extern short *ztl;
extern unsigned char *stats_flag;
extern unsigned int *stomp_flag;
extern unsigned int *stomp_ztl;
extern unsigned char *byte_stream;
extern unsigned char *tbuf;
extern unsigned short *list;

```



```

extern int No_of_chars;
extern int EOF_symbol; /* Index of EOF symbol */
extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */
extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short int cum_freq[Max_No_of_symbols+1];
extern int buffer_index;
extern unsigned short bit_index;
extern unsigned short T_BYTES;
extern short maxval;

extern unsigned short *ALLOC;
extern int FIRST_ENC, FCNT_ENC; /* 1st pass bite size */
extern unsigned char *passes_d;

/* Required on-line storage for high speed */

extern int STOP;
extern int BYTE_CNT;
extern unsigned short THRESH;
extern int BITE;
extern int SIG_COEF;

extern int whoami(); /* Function will return the PP number */

void start_model(int nchars);
void update_model(int symbol);
void encode_symbol(int symbol);
void bit_plus_follow(int bit);
void output_bit(int bit);
void new_dbits(int m, int c, int number, int s);
void new_dbits2(int m, int s);
void comp_ztr(int s);
void subpass(int t);

/*****
*
*   Function : start_encoding
*   Args      : none
*
*   Description:
*       start_encoding will be called to initialize the
*       arithmetic coder
*
*   Return Values:
*       None
*
*****/

void start_encoding()
{
    low = 0; /* Full code range */
    high = Top_value;
    bits_to_follow = 0; /* No bits to follow */
}

```

```

/*****
*
*   Function : start_outputing_bits
*   Args      : none
*
*   Description:
*       start_outputing_bits will be called to initialize the
*       the bit buffer
*
*   Return Values:
*       None
*
*****/

```

```

void start_outputing_bits()
{
    buffer = 0;          /* Buffer is empty at start */
    bits_to_go = 8;
}

```

```

/*****
*
*   Function : input_block
*   Args      : s - subblock number (0 - 5)
*               p - pair number (0,1)
*
*   Description:
*       input_block will be called to copy the coefficients from
*       one section of memory in in-place format to another
*       place in memory in zero-tree format
*
*   Return Values:
*       None
*
*****/

```

```

void input_block(s,p)
    int s,p;

{
    int i,j,k,l;

    i = s * 256;
    for(k=NSCALES-2;k>0;k--)
    {
        if (k==(NSCALES-2))
            for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=0;l<(1<<NSCALES);l+=(2<<k))
                    stats_val[i++] = coeff_block[p*512 + j*32+l];

            for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
                    stats_val[i++] = coeff_block[p*512 + j*32+l];

            for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=0;l<(1<<NSCALES);l+=(2<<k))
                    stats_val[i++] = coeff_block[p*512 + j*32+l];

            for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
                    stats_val[i++] = coeff_block[p*512 + j*32+l];
    }
}

```

```

/*****
*
*   Function : comp_dbits
*   Args      : m - index of coefficient
*                c1- first child index
*                c2- second child index
*                c3- third child index
*                s - subblock number (0 - 5)
*
*   Description:
*       comp_dbits will be called to determine if the root of the
*       zero tree pass the threshold or not, if it does then it
*       is tagged as significant, its value is modified. Then its
*       children are checked to see if any coefficients in its
*       line are above the threshold, if they aren't then a
*       passdown flag is given to the children - to signify no
*       significance under them.
*
*   Return Values:
*       None
*
*****/

void comp_dbits(m,c1,c2,c3,s)
    int m,c1,c2,c3,s;
{
    int sym,tflg,t,m0;

    m0 = m;
    m = m + (s * 256);
    if ((stats_flag[m]&6)==0)
    {
        tflg = ((TMASK&ztl[m0 + (s * 64)]) == 0);
        t = (TMASK&abs(stats_val[m]))>>SHFT_DN;
        sym = t + (((t!=0)&&(stats_val[m]>0)) ? ((1<<BITE)-1):0) + ((~tflg)&1);

        if ((t>0)&&(list_index<254))
            list[list_index++] = m;

        sym_array[sym_index++] = sym;

        if (sym_index > maxsyms)
        {
            do_syms();
            sym_index = 0;
        }

        stats_val[m] = (t>0) ? (abs(stats_val[m]) - (((t*THRESH)>>(BITE-1))+(THRESH>>BITE))) :
stats_val[m];

        if (tflg!=0)
        {
            pruned_children[s] = 1;
            stats_flag[(s*256)+c1] = stats_flag[(s*256)+c1] | (tflg<<1);
            stats_flag[(s*256)+c2] = stats_flag[(s*256)+c2] | (tflg<<1);
            stats_flag[(s*256)+c3] = stats_flag[(s*256)+c3] | (tflg<<1);
        }

        stats_flag[m] = (t!=0)<<2;
    }
    else if ((stats_flag[m]&2)!=0)
    {
        stats_flag[m] = stats_flag[m] & 252;
        stats_flag[(s*256)+c1] = stats_flag[(s*256)+c1] | 2;
        stats_flag[(s*256)+c2] = stats_flag[(s*256)+c2] | 2;
        stats_flag[(s*256)+c3] = stats_flag[(s*256)+c3] | 2;
        pruned_children[s] = 1;
    }
}

```

```

}

/*****
*
*   Function : p_code
*   Args      : none
*
*   Description:
*       p_code will be called to coordinate the generation of the
*       bitstream from 6 sets of subblocks
*
*   Return Values:
*       None
*
*****/

void p_code()
{
    int k,l,i,j,mask,bite,pmin;

    sym_index = 0;

    /* wait for coefficients to come in to determine maxval */

    maxval = 0;

    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    /* Calculate local maximum value of coefficients */

    for (k=0;k<128;k++)
        if (abs(stats_val[k]) > maxval)
            maxval = abs(stats_val[k]);

    local_maxval[whoami()] = maxval;

    /* tell MP done with maxval calculation */

    asm("    x2 = 0x00002100");
    asm("    cmd = x2");

    /* wait for global maxval computation to complete */

    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    maxval = 1<<global_maxval;

    /* Determine number of bitplanes to process on first and */
    /* successive passes */

    FIRST_ENC = tbuf[0];

    if (FIRST_ENC == 6)
    {
        bite = 3;
        RB_ENC = 3;
    }
    else if (FIRST_ENC == 5)
    {
        bite = 3;
        RB_ENC = 2;
    }
    else if (FIRST_ENC==4)
    {
        bite = 2;
        RB_ENC = 2;
    }
}

```

```

else
{
    bite = FIRST_ENC;
    RE_ENC = 1;
}

/* Compute mask */

mask = maxval;
for(k=1;k<bite;k++)
    mask = mask | (maxval>>k);

pmin = 100;

/* tell MP this PP is ready to encode the bit stream */

asm("    x2 = 0x00002100");
asm("    cmnd = x2");

/* Main Loop: Continue until stop condition is reached */

while (*pp_stop_encode == 0)
{
    BITE = bite;
    PASS_ENC = 0;
    STOP = 0;
    BYTE_CNT = 0;

    THRESH = maxval;
    SHFT_DN = global_maxval - BITE + 1;
    TMASK = mask;
    buffer_index = 0;

    list_index = 0;

    start_model(1<<(BITE+1));
    start_outputting_bits();
    start_encoding();

    /* wait for new coefficients */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    if (*pp_stop_encode == 0)
    {
        for (i=0;i<6;i=i+2)
        {
            input_block(i,0);
            comp_ztr(i);

            input_block(i + 1,1);
            comp_ztr(i + 1);

            if (i!=4)
            {
                i = i + 1;
                i = i - 1;

                /* tell MP done inputting this block of coefficients */
                asm("    x2 = 0x00002100");
                asm("    cmnd = x2");

                /* wait for new coefficients */
                while((INTFLG&(1<<20))==0);
                INTFLG = 1<<20;
            }
        }
    }
}

```

```

T_BYTES = ALLOC[0];
for (bit_index=0;bit_index<T_BYTES;bit_index++) byte_stream[bit_index] = 0;
bit_index = 0;

for(j=0;j<384;j++)
    stomp_flag[j] = 0;

while (STOP == 0)
{
    code_dlist2();
}

if (ALLOC[0] <3)
    passes_d[0] = 0;
else if (ALLOC[0]==3)
    passes_d[0] = PASS_ENC-4;
else if (ALLOC[0]<7)
    passes_d[0] = PASS_ENC-2;
else
{
    passes_d[0] = PASS_ENC-1;
    if (PASS_ENC-1 < pmin)
        pmin = PASS_ENC-1;
}

/* tell MP done with this block of pixels */

asm("    x2 = 0x00002100");
asm("    cmnd = x2");

} /*end if */
} /* end while */

/* Inform MP of number of passes at this BITE size */

if (pmin==0)
    FIRST_ENC -= 1;
else if ((pmin > FIRST_ENC)&&(FCNT_ENC > 2)&&(pmin<7))
{
    FCNT_ENC = 0;
    FIRST_ENC = pmin-1;
}
else if (pmin > FIRST_ENC)
    FCNT_ENC++;
else if (pmin<=FIRST_ENC)
{
    FIRST_ENC = pmin;
    FCNT_ENC = 0;
}

tbuf[1] = FIRST_ENC;
}

```

```

/*****
**
** main.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Main PP Program that calls all the other routines to perform the wavelet
** decomposition and bitstream generation.
**
*****/

#include "modlp.h"

#define NSCALES 4

    unsigned short T_BYTES;
    unsigned char *pic;
    short *img;

    short *stats_val;
    short *stats_apx;
    unsigned char *stats_flag;
    unsigned int *stomp_flag;
    unsigned int *stomp_ztl;
    unsigned char *byte_stream;
    short *coeff_block;
    unsigned short *ztl;
    unsigned char *tbuf;
    unsigned short *ALLOC;

    unsigned short *list;

int No_of_chars;

int EOF_symbol; /* Index of EOF symbol */

int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

unsigned char char_to_index[Max_No_of_chars]; /* JCW old version was int */
unsigned char index_to_char[Max_No_of_symbols+1]; /* JCW old version was int */

short int cum_freq[Max_No_of_symbols+1]; /* JCW old version was int */

int buffer_index;

extern void subdec_vert(int outerloop,int innerloop);
extern void subdec_horiz(int outerloop,int innerloop);

extern int calc_n_sub_mean(int oldmean);

extern int whoami();

/* ***** MAIN PROGRAM ***** */

register extern volatile unsigned int INTFLG;

    short column_outerloop[5] = { 8, 16, 32, 16, 8 };
    short row_outerloop[5] = { 4, 8, 16, 8, 4 };

    short column_innerloop[5] = { 120, 60, 30, 15, 8 };
    short row_innerloop[5] = { 256, 128, 64, 32, 16 };

    unsigned char vert_loop_index[6] = { 16, 4, 1, 1, 1 };
    unsigned char horiz_loop_index[6] = { 15, 5, 1, 1, 1 };

    short maxval;

```

```

extern shared int mean_pp[4];
extern shared int global_mean;

unsigned char *passes_d;

extern unsigned char PASS_ENC,PASS_DEC;

int STOP;
int BYTE_CNT;
unsigned short THRESH;
int BITE;
int SIG_COEF;

int FIRST_ENC,FIRST_DEC;
int FCNT_ENC,FCNT_DEC;

unsigned int getaway_address;
unsigned char quick_getaway = 0;

main()
{
    int k,l;
    int mean;

    /* initialize pointers */

    asm("    dl = &(dba)");
    asm("    *(xba+$pic) = dl");
    asm("    *(xba+$img) = dl");
    asm("    *(xba+$stats_val) = dl");
    asm("    *(xba+$stats_apx) = dl");
    asm("    dl = &(dba + 0x8000)");
    asm("    *(xba+$stats_flag) = dl");
    asm("    *(xba+$stomp_flag) = dl");
    asm("    *(xba+$coeff_block) = dl");
    asm("    x0 = 0x8602");
    asm("    nop");
    asm("    dl = &(dba + x0)");
    asm("    *(xba+$list) = dl");
    asm("    dl = &(dba + 0xc00)");
    asm("    *(xba+$ztl) = dl");
    asm("    *(xba+$stomp_ztl) = dl");
    asm("    dl = &(pba + 0x630)");
    asm("    *(xba+$byte_stream) = dl");
    asm("    dl = &(pba + 0x620)");
    asm("    *(xba+$passes_d) = dl");
    asm("    dl = &(pba + 0x5fc)");
    asm("    *(xba+$tbuf) = dl");
    asm("    dl = &(pba + 0x600)");
    asm("    *(xba+$ALLOC) = dl");

    FIRST_ENC = 4;
    FIRST_DEC = 4;

    FCNT_ENC = 0;
    FCNT_DEC = 0;

    /* Clear the message interrupt flag that comes from the MP, just in case */

    INTFLG = 1<<20;
    quick_getaway = 0;

    while (1)
    {
        mean = 0;

```



```

/* Decompose image */
for(k=0;k<(NSCALES+1);k++)
{
    for(l=0;l<horiz_loop_index[k];l++)
    {
        /* wait for new pixels */

        while((INTFLG&(1<<20))==0);

        INTFLG = 1<<20;

        /* if we are on the first pass that means we have 8 bit
           pixels that have come in that need to be zero meaned
           and converted to 16 bit signed numbers */

        if (k==0)
            mean = mean + calc_n_sub_mean(global_mean);

        subdec_horiz(row_outerloop[k],row_innerloop[k]);

        /* tell MP done with this block of pixels */

        asm("    d7 = 0x00002100");
        asm("    cmnd = d7");

    }

    /* Put out local mean for MP to do global mean calculation */

    mean_pp[whoami()] = mean;

    if (k!=NSCALES)
        for(l=0;l<vert_loop_index[k];l++)
        {
            /* wait for new pixels */

            while((INTFLG&(1<<20))==0);

            INTFLG = 1<<20;

            subdec_vert(column_outerloop[k],column_innerloop[k]);

            /* tell MP done with this block of pixels */

            asm("    d7 = 0x00002100");
            asm("    cmnd = d7");

        }

    }

    /* Generate the bitstream */

    p_code();
}
}

```

```

/*****
**
** mean2.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $scal_n_sub_mean
** $whoami
**
*****/

        .global $scal_n_sub_mean
        .global $whoami

/*****
*
* Function : $scal_n_sub_mean
* Args    : oldmean
* Passed in : d1
*
* Description:
*   oldmean - the mean on the previous image
*
*   $scal_n_sub_mean will be called to both calculate the sum
*   of the pixels in this patch, and to subtract off the
*   previous images mean (performing an 8 bit minus and 16
*   bit subtraction, yielding a 16 bit result, which is then
*   further processed by shifting it to the left 2 places.
*
* Return Values:
*   d5 - returns the sum of this patch of pixels.
*
*****/

$scal_n_sub_mean:
        d5 = 0
        a8 = &*(dba) + 4094
        a0 = &*(dba) + 2047

        d1 = 0 - (d1 << 2)          ; make mean so it can be subtracted

        le2 = loopend
        lrs2 = 510

        d2 =ub *a0--
        d5 = d5 + d2

        x1 = d1 + (d2 << 2)
        || d2 =ub *a0--
        *a8-- =h x1
        || d5 = d5 + d2

        x1 = d1 + (d2 << 2)
        || d2 =ub *a0--
        *a8-- =h x1
        || d5 = d5 + d2

        x1 = d1 + (d2 << 2)
        || d2 =ub *a0--
        *a8-- =h x1
        || d5 = d5 + d2

        x1 = d1 + (d2 << 2)
        || d2 =ub *a0--
loopend:
        *a8-- =h x1

```

# NAWCWD TP 8442

```

|| d5 = d5 + d2

    x1 = d1 + (d2 << 2)
|| d2 =ub  *a0--
    *a8-- =h  x1
|| d5 = d5 + d2

    x1 = d1 + (d2 << 2)
|| d2 =ub  *a0--
    *a8-- =h  x1
|| d5 = d5 + d2

    x1 = d1 + (d2 << 2)
|| d2 =ub  *a0--
    *a8-- =h  x1
|| d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    *a8-- =h  x1

br = iprs
nop
nop

```

```

/*****
*
*   Function   : $whoami
*   Args      : none
*   Passed in :
*
*   Description:
*       $whoami will be called to determine which PP this is.
*
*   Return Values:
*       d5 - returns the PP number assigned to this PP.
*
*****/

```

```

$whoami:
    d5 = comm & 0x03
    br = iprs
    nop
    nop

```

```

/*****
**
** modlp.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains start_model subroutine. This subroutine initializes
** the translation tables and the frequency counters for the
** arithmetic coder.
**
*****/

#include "modlp.h"
#include "arith.h"

unsigned short bit_index;

short int freq[Max_No_of_symbols+1]; /* Symbol frequencies */

extern int No_of_chars;
extern int EOF_symbol; /* Index of EOF symbol */
extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short int cum_freq[Max_No_of_symbols+1];
extern int low,high;
extern short bits_to_follow;
extern unsigned char bits_to_go;
extern unsigned char buffer;
extern int BYTE_CNT;
extern unsigned short T_BYTES;
extern unsigned char *byte_stream;
extern int buffer_index;
extern int STOP;

/*****
*
* Function : start_model
* Args : int nchars
*
* Description:
* nchars - the number of symbols to be encoded by the model
*
* start_model will be called when the number of characters
* to be encoded by the model changes. It re-initializes
* the tables used by the encoder.
*
* Return Values:
* None
*
*****/

void start_model(nchars)
int nchars;
{
int i;

/* Initialize number of chars */

No_of_chars = nchars;
No_of_symbols = nchars;

/* Setup translation tables */

```

```
for(i=0;i<No_of_chars;i++)
{
    char_to_index[i] = i+1;
    index_to_char[i+1] = i;
}

/* Initialize frequency counts */

for(i=0;i<=No_of_symbols;i++)
{
    freq[i] = 1;
    cum_freq[i] = No_of_symbols-i;
}
freq[0] = 0;
}
```

## NAWCWD TP 8442

```
/* DECLARATIONS FOR ARITHMETIC CODING AND DECODING */

/* Size of arithmetic code values */

#define Code_value_bits 9 /* Number of bits in a code value */
typedef short code_value; /* Type of arithmetic code value */

#define Top_value (((long)1<<Code_value_bits)-1) /* Largest code val */

/* Half and Quarter points in code value range */

#define First_qtr (Top_value/4+1) /* Points after first quarter */
#define Half      (2*First_qtr)   /* Points after first half */
#define Third_qtr (3*First_qtr)   /* Points after third quarter */
```

```

/*****
**
** mp.c (MP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** MP Program that orchestrates data movement and kicks off PP's to
** implement the balanced wavelet algorithm written by Chuck Creusere.
** This version is meant to be run from the PC host.
**
*****/

#include <stdlib.h>
#include <stdio.h>
#include <mvp.h>

#include <mvp_hw.h>
#include <mp_ptreq.h>

#include "icl.h"
#include "vil24.h"
#include "vol.h"
#include "bgl.h"
#include "pcic80.h"
#include "cil.h"

/* define compression ratio.  Ratio is actually #:1 compression */
#define compression 40

#define HOST

#define CAMERA

#define ENCODE
#define ENCODE_STREAM

#define DISPLAY
#define SHOWDISPLAY

#define headersize 0

#define XSIZE 512
#define YSIZE 240

/*Number of Subblocks in the X direction */
#define AX 16

/*Number of Subblocks in the Y direction */
#define AY 15

/*
** Define a bit mask for host request bit
*/
#define SigBit(x)      (((UINT32)1)<<(x))
#define HostRequestBitMask SigBit(8)
#define FrameDoneBit 8

/*****
/* A pointer to the array of pixels for the number characters */
extern int *number_pixels;

/* Place certain variables in MP Parameter RAM for the sake of speed */

#pragma DATA_SECTION(Semaphore,"mp_vars")
long Semaphore;

#pragma DATA_SECTION(encode_time,"mp_vars")
#pragma DATA_SECTION(decode_time,"mp_vars")

```

```

int encode_time, decode_time, proc_time;

#pragma DATA_SECTION(start_time, "mp_vars")
unsigned int start_time;

#pragma DATA_SECTION(local_alloc, "mp_vars")
unsigned short local_alloc[AX*AY/6];

#pragma DATA_SECTION(comp_table, "mp_vars")
unsigned char comp_table[5] = {10, 20, 40, 80, 100};

#pragma DATA_SECTION(last_comp, "mp_vars")
unsigned char last_comp = 2;

#pragma DATA_SECTION(compression_ratio, "mp_vars")
UINT32 *compression_ratio;

/* Place shared variables in PP Parameter RAM where both the PPs and MP can get to them
*/

#pragma DATA_SECTION(mean_pp, "sh_vars")
#pragma DATA_SECTION(global_mean, "sh_vars")

    shared int mean_pp[4];
    shared int global_mean;

#pragma DATA_SECTION(local_maxval, "sh_vars")
#pragma DATA_SECTION(global_maxval, "sh_vars")

    shared int local_maxval[4];
    shared int global_maxval;

static void SignalHandler(UINT32 Signals);
static void init_alloc_table(UINT32 index);
static void init_alloc_table2(UINT32 index);

/*****/

void task (void *arg)
{
    unsigned char times=0;
    unsigned char flip = 0;
    unsigned int stream_addr[2];
    PCIC80STAT ReturnVal;

    FVIL24 pVil24;
    ICL_IMG *vim,*f0,*f1;
    unsigned int dx,dy;
    int i,j;
    int lp;
    unsigned char vert_loop_index[6],horiz_loop_index[6];
    int jump_col[6];
    long jump_row[6];

    long *unused_pixels;

    long *ptr;          /* temp pointer          */
    PTREQ *p[10];       /* temp pointer to packet transfer structure */

    int temp_maxval;

/* JCWtest */

    char string1[32];
    int temp_str;
    float temp_time;

```



```

int junki;
unsigned short junkfill = 0;
unsigned short * fillme = (unsigned short *) 0x90320000;

int k,l;
long templ;
unsigned char tbuf;

int t_alloc;
int p_alloc;
int r_alloc;

unsigned char * tbuf_pp0      = (unsigned char *) 0x010005fc;
unsigned char * tbuf_pp1      = (unsigned char *) 0x010015fc;
unsigned char * tbuf_pp2      = (unsigned char *) 0x010025fc;
unsigned char * tbuf_pp3      = (unsigned char *) 0x010035fc;

unsigned char min_first;

unsigned char ppnexttask[4];
unsigned short ppalloc[4];
unsigned char current_block;
unsigned int table_pointer = 0x80380000;
unsigned int table_address[4];
unsigned short table_size[4];
unsigned char pprequesting;
unsigned char ppinfo[4];
unsigned char ppdone;

unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

unsigned char DONE1;

UINT32 temp_ulong;

Semaphore = 0;
comp_table[0] = 10;
comp_table[1] = 20;
comp_table[2] = 40;
comp_table[3] = 80;
comp_table[4] = 100;
last_comp = 2;
compression_ratio = (UINT32 *) 0x010107D0;

stream_addr[0] = 0x90023000;
stream_addr[1] = 0x90028000;

NOCACHE_INT(number_pixels[0]) = 0x80;

NOCACHE_INT(global_mean) = 0x5e;

*pp_stop_encode = 0;

/* initialize FIRST variable on all PPs - used for first pass # of bitplanes to process
*/
tbuf_pp0[0] = 4;
tbuf_pp1[0] = 4;
tbuf_pp2[0] = 4;
tbuf_pp3[0] = 4;

#ifdef HOST
    ReturnVal = CilSigHandler(SignalHandler);
    if (ReturnVal != CIL_OK)
    {
        /*** Cannot register signal handler ***/
    }
    while(1);
    return;
}
#endif

```

```

dx = 512;                /* size of ROI which will be initialised*/
dy = 480;                /* if too big, may not run at frame rate*/

pVil24 = Vil24Open();      /* Open VIL module */
if ( pVil24 == NULL ) exit(0); /* Failed to open VIL module */
Vil24Initialize(pVil24,VIL24_EIA_DEFAULT); /* set up for CCIR camera */
Vil24SetVcrMode(pVil24,1); /* ensure lock to poor sources */
Vil24SetROI(pVil24,64,0,dx,dy); /* set ROI in the center */

vim = IclCreateHdr(1,1,ICL_IMG_CUSTOM); /* create image header structure ... */
Vil24InitImgHdr(pVil24,vim); /* ... and initialise to describe VIL */

VolSetDisplay(VOL_VGA8_1024); /* setup colour display for VIM-8 */
VolSetGreyLUT();

p[0] = (PTREQ *) (MP_PARM_RAM + 0x2c0);
p[1] = p[0] + 1;
p[2] = p[0] + 2;
p[3] = p[0] + 3;
p[4] = p[0] + 4;
p[5] = p[0] + 5;
p[6] = p[0] + 6;
p[7] = p[0] + 7;
p[8] = p[0] + 8;
p[9] = p[0] + 9;

/* Set MP list pointer to point to first PT */
ptr = (long *) (MP_PTREQ_PTR);
*ptr = (long) p[0];

/* Fifo Bank 0 -> DRAM (8 bit data)*/

p[0]->link = p[0]; /* point to this PT */
p[0]->word[0] = 0x80000000; /* Weird Fifo to linear */
p[0]->word[1] = 0xa0000001; /* Src address is VIM pixel fifo */
p[0]->word[2] = 0x80300000; /* Dst address is DRAM */
p[0]->word[3] = 0x01ff0001; /* Src B count Src A count */
p[0]->word[4] = 0x00ef0200; /* Dst B count Dst A count */
p[0]->word[5] = 0xef; /* Src C count */
p[0]->word[6] = 0; /* Dst C count */
p[0]->word[7] = 0x08; /* Src B pitch */
p[0]->word[8] = 0x200; /* Dst B pitch */
p[0]->word[9] = 0x400; /* Src C pitch */
p[0]->word[10] = 0; /* Dst C pitch */
p[0]->word[11] = 0; /* Src transparency upper */
p[0]->word[12] = 0; /* Src transparency lower */
p[0]->word[13] = 0; /* Reserved */
p[0]->word[14] = 0; /* Reserved */

/* SDRAM -> internal coefficient blocks (16 bit data) (32x16 words)*/

p[1]->link = p[1]; /* point to this PT */
p[1]->word[0] = 0x80000000; /* Contig. Mem to int no update */
p[1]->word[1] = 0x80320000; /* Src address is SDRAM */
p[1]->word[2] = 0x00008000; /* Dst address is internal */
p[1]->word[3] = 0x000f0040; /* Src B count Src A count */
p[1]->word[4] = 0x00000400; /* Dst B count Dst A count */
p[1]->word[5] = 0x00; /* Src C count */
p[1]->word[6] = 0; /* Dst C count */
p[1]->word[7] = 0x0400; /* Src B pitch */
p[1]->word[8] = 0x000; /* Dst B pitch */
p[1]->word[9] = 0x40; /* Src C pitch */
p[1]->word[10] = 0x1000; /* Dst C pitch */
p[1]->word[11] = 0; /* Src transparency upper */
p[1]->word[12] = 0; /* Src transparency lower */
p[1]->word[13] = 0; /* Reserved */
p[1]->word[14] = 0; /* Reserved */

```

# NAWCWD TP 8442

/\* SDRAM (8 bit data) -> VRAM (raw image) \*/

```
p[2]->link = p[2];          /* point to this PT          */
p[2]->word[0] = 0x80000000;   /* linear to VRAM          */
p[2]->word[1] = 0x80300000;   /* Src address is SDRAM    */
p[2]->word[2] = 0xb4000000;   /* Dst address is VRAM     */
p[2]->word[3] = 0x00038000;   /* Src B count Src A count */
p[2]->word[4] = 0x00ff0200;   /* Dst B count Dst A count */
p[2]->word[5] = 0x00;        /* Src C count             */
p[2]->word[6] = 0;           /* Dst C count             */
p[2]->word[7] = 0x8000;      /* Src B pitch             */
p[2]->word[8] = 0x800;       /* Dst B pitch             */
p[2]->word[9] = 0x00;        /* Src C pitch             */
p[2]->word[10] = 0x0000;     /* Dst C pitch             */
p[2]->word[11] = 0;         /* Src transparency upper   */
p[2]->word[12] = 0;         /* Src transparency lower   */
p[2]->word[13] = 0;         /* Reserved                */
p[2]->word[14] = 0;         /* Reserved                */
```

/\* DRAM (rows and 8 bit data) -> internal RAM \*/

/\* Can be used 4 times then change dst <- this can be done 8 times \*/

```
p[3]->link = p[3];          /* point to this PT          */
p[3]->word[0] = 0x80000202;   /* Contig. Mem to int w/s update */
p[3]->word[1] = 0x80300000;   /* Src address is DRAM      */
p[3]->word[2] = 0x00000000;   /* Dst address is internal   */
p[3]->word[3] = 0x00000800;   /* Src B count Src A count */
p[3]->word[4] = 0x00000800;   /* Dst B count Dst A count */
p[3]->word[5] = 0x00;        /* Src C count             */
p[3]->word[6] = 0;           /* Dst C count             */
p[3]->word[7] = 0x0000;      /* Src B pitch             */
p[3]->word[8] = 0x000;       /* Dst B pitch             */
p[3]->word[9] = 0x0800;      /* Src C pitch             */
p[3]->word[10] = 0x1000;     /* Dst C pitch             */
p[3]->word[11] = 0;         /* Src transparency upper   */
p[3]->word[12] = 0;         /* Src transparency lower   */
p[3]->word[13] = 0;         /* Reserved                */
p[3]->word[14] = 0;         /* Reserved                */
```

/\* DRAM (columns and 16 bit data) -> internal RAM \*/

/\* Can be used 4 times then change dst <- this can be done 8 times \*/

```
p[4]->link = p[4];          /* point to this PT          */
p[4]->word[0] = 0x80000002;   /* Contig. Mem to int w/dst updt */
p[4]->word[1] = 0x80320000;   /* Src address is DRAM      */
p[4]->word[2] = 0x00000000;   /* Dst address is internal   */
p[4]->word[3] = 0x00ff0010;   /* Src B count Src A count */
p[4]->word[4] = 0x00001000;   /* Dst B count Dst A count */
p[4]->word[5] = 0x00;        /* Src C count             */
p[4]->word[6] = 0;           /* Dst C count             */
p[4]->word[7] = 0x0400;      /* Src B pitch             */
p[4]->word[8] = 0x000;       /* Dst B pitch             */
p[4]->word[9] = 0x10;        /* Src C pitch             */
p[4]->word[10] = 0x1000;     /* Dst C pitch             */
p[4]->word[11] = 0;         /* Src transparency upper   */
p[4]->word[12] = 0;         /* Src transparency lower   */
p[4]->word[13] = 0;         /* Reserved                */
p[4]->word[14] = 0;         /* Reserved                */
```

/\* internal RAM -> DRAM (columns and 16 bit data) \*/

/\* Can be used 4 times then change src then this can be repeated 8 times \*/

```
p[5]->link = p[5];          /* point to this PT          */
p[5]->word[0] = 0x80000200;   /* int to Contig. Mem w/src updt */
p[5]->word[1] = 0x00000000;   /* Src address is internal   */
p[5]->word[2] = 0x80320000;   /* Dst address is DRAM      */
p[5]->word[3] = 0x00001000;   /* Src B count Src A count */
p[5]->word[4] = 0x00ff0010;   /* Dst B count Dst A count */
```

# NAWCWD TP 8442

```

p[5]->word[5] = 0x00;          /* Src C count          */
p[5]->word[6] = 0;             /* Dst C count          */
p[5]->word[7] = 0x0000;        /* Src B pitch          */
p[5]->word[8] = 0x400;         /* Dst B pitch          */
p[5]->word[9] = 0x1000;        /* Src C pitch          */
p[5]->word[10] = 0x0010;       /* Dst C pitch          */
p[5]->word[11] = 0;            /* Src transparency upper */
p[5]->word[12] = 0;            /* Src transparency lower */
p[5]->word[13] = 0;            /* Reserved              */
p[5]->word[14] = 0;            /* Reserved              */

/* DRAM (rows and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst then this can be repeated 8 times */

p[6]->link = p[6];             /* point to this PT      */
p[6]->word[0] = 0x80000002;     /* Contig. Mem to int w/dst updt */
p[6]->word[1] = 0x80320000;     /* Src address is DRAM    */
p[6]->word[2] = 0x00000000;     /* Dst address is internal */
p[6]->word[3] = 0x00001000;     /* Src B count Src A count */
p[6]->word[4] = 0x00001000;     /* Dst B count Dst A count */
p[6]->word[5] = 0x00;          /* Src C count          */
p[6]->word[6] = 0;             /* Dst C count          */
p[6]->word[7] = 0x0000;        /* Src B pitch          */
p[6]->word[8] = 0x000;         /* Dst B pitch          */
p[6]->word[9] = 0x0000;        /* Src C pitch          */
p[6]->word[10] = 0x1000;       /* Dst C pitch          */
p[6]->word[11] = 0;            /* Src transparency upper */
p[6]->word[12] = 0;            /* Src transparency lower */
p[6]->word[13] = 0;            /* Reserved              */
p[6]->word[14] = 0;            /* Reserved              */

/* internal RAM -> DRAM (rows and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[7]->link = p[7];             /* point to this PT      */
p[7]->word[0] = 0x80000200;     /* int to DRAM w/src updt */
p[7]->word[1] = 0x00000000;     /* Src address is internal */
p[7]->word[2] = 0x80320000;     /* Dst address is DRAM    */
p[7]->word[3] = 0x00001000;     /* Src B count Src A count - */
p[7]->word[4] = 0x00001000;     /* Dst B count Dst A count */
p[7]->word[5] = 0x00;          /* Src C count          */
p[7]->word[6] = 0;             /* Dst C count          */
p[7]->word[7] = 0x0000;        /* Src B pitch          */
p[7]->word[8] = 0x000;         /* Dst B pitch          */
p[7]->word[9] = 0x1000;        /* Src C pitch          */
p[7]->word[10] = 0x0000;       /* Dst C pitch          */
p[7]->word[11] = 0;            /* Src transparency upper */
p[7]->word[12] = 0;            /* Src transparency lower */
p[7]->word[13] = 0;            /* Reserved              */
p[7]->word[14] = 0;            /* Reserved              */

/* internal RAM -> VRAM (rows and 8 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[8]->link = p[8];             /* point to this PT      */
p[8]->word[0] = 0x80000202;     /* int to VRAM w/dst updt */
p[8]->word[1] = 0x00000000;     /* Src address is internal */
p[8]->word[2] = 0xb4000200;     /* Dst address is VRAM    */
p[8]->word[3] = 0x00000800;     /* Src B count Src A count */
p[8]->word[4] = 0x00030200;     /* Dst B count Dst A count */
p[8]->word[5] = 0x00;          /* Src C count          */
p[8]->word[6] = 0;             /* Dst C count          */
p[8]->word[7] = 0x0000;        /* Src B pitch          */
p[8]->word[8] = 0x400;         /* Dst B pitch          */
p[8]->word[9] = 0x1000;        /* Src C pitch          */
p[8]->word[10] = 0x1000;       /* Dst C pitch          */
p[8]->word[11] = 0;            /* Src transparency upper */
p[8]->word[12] = 0;            /* Src transparency lower */
p[8]->word[13] = 0;            /* Reserved              */

```

# NAWCWD TP 8442

```

p[8]->word[14] = 0;          /* Reserved */

/* internal RAM -> SDRAM (byte stream 8 bit data) */

p[9]->link = p[9];           /* point to this PT */
p[9]->word[0] = 0x80000202;    /* int to VRAM w/dst updt */
p[9]->word[1] = 0x01000630;    /* Src address is internal */
p[9]->word[2] = 0x90023000;    /* Dst address is Shared DRAM */
p[9]->word[3] = 0;            /* Src B count Src A count */
p[9]->word[4] = 0;            /* Dst B count Dst A count */
p[9]->word[5] = 0x00;         /* Src C count */
p[9]->word[6] = 0;            /* Dst C count */
p[9]->word[7] = 0x0000;       /* Src B pitch */
p[9]->word[8] = 0x000;        /* Dst B pitch */
p[9]->word[9] = 0x1000;       /* Src C pitch */
p[9]->word[10] = 0;           /* Dst C pitch */
p[9]->word[11] = 0;           /* Src transparency upper */
p[9]->word[12] = 0;           /* Src transparency lower */
p[9]->word[13] = 0;           /* Reserved */
p[9]->word[14] = 0;           /* Reserved */

Vil24SetSequenceMode(pVil24,1); /* set VIL to sequence mode */
Vil24StartCapture(pVil24);      /* start capturing images */

IE = 0x01;

command(0x2000000f);           /* unhalt PP0 */

vert_loop_index[0] = 16;
vert_loop_index[1] = 4;
vert_loop_index[2] = 1;
vert_loop_index[3] = 1;
vert_loop_index[4] = 1;

horiz_loop_index[0] = 15;
horiz_loop_index[1] = 5;
horiz_loop_index[2] = 1;
horiz_loop_index[3] = 1;
horiz_loop_index[4] = 1;

jump_col[0] = 16;
jump_col[1] = 64;
jump_col[2] = 256;
jump_col[3] = 256;
jump_col[4] = 256;

jump_row[0] = 0x1000; /* not used */
jump_row[1] = 0x3000;
jump_row[2] = 0xF000;
jump_row[3] = 0x10000;
jump_row[4] = 0x10000;

/* Zero out unused lines in the input (lines 240-255) */

unused_pixels = (long *) 0x8031e000;

for (i=0;i<2048;i++)
    NOCACHE_INT(unused_pixels[i]) = 0;

/* Initialize the adaptive bit allocation tables on the PPs */

init_alloc_table(2);

Vil24SetReadBank(pVil24,0);    /* prepare to read 1st field (even lines) */

/*****

while (1)
{

```

```

#ifdef CAMERA
    Vil24WaitForData(pVil24);          /* wait for image data      */
    Vil24SetReadBank(pVil24,0);        /* prepare to read 1st field */
#endif

encode_time = 0xffffffff - TCOUNT;
TCOUNT = 0xffffffff;

#ifdef CAMERA
    *ptr = (long) p[0];

    /* kick off TC to transfer pixel info to SDRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);
#endif

/* clear the interrupt flag */
INTPEN = 0xf0000;

#ifdef HOST

/* Read Compression Ratio */

CilReadMailbox(1, (PUINT32)compression_ratio);

if (*compression_ratio > 4) *compression_ratio = 2;
if (*compression_ratio < 0) *compression_ratio = 2;

if (*compression_ratio != last_comp)
{
    last_comp = *compression_ratio;
    init_alloc_table(last_comp);
}
#endif

/* Display the incoming raw data (doubling the rows) */

p[2]->word[2] = 0xb4000000;

*ptr = (long) p[2];

/* kick off TC to transfer pixel from SDRAM to VRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

p[2]->word[2] = 0xb4000400;

/* kick off TC to transfer pixel from SDRAM to VRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

/* Encode coefficients of the raw image */

#ifdef ENCODE

/* TCOUNT = 0xffffffff; */

p[3]->word[1] = 0x80300000;          /* Src address is SDRAM      */

for (lp=0; lp<5; lp++)
{
    /* Do 32 rows */

    for (i=0; i<horiz_loop_index[lp]; i++)
    {
        if (i==0)
        {
            p[6]->word[1] = 0x80320000;          /* Src address is SDRAM      */
            p[7]->word[2] = 0x80320000;          /* Dst address is SDRAM      */

```

```

    }

    p[3]->word[2] = 0x00000000;      /* Dst address is internal */
    p[6]->word[2] = 0x00000000;      /* Dst address is internal */
    p[7]->word[1] = 0x00000000;      /* Src address is internal */

    if (lp==0)
        *ptr = (long) p[3];
    else
        *ptr = (long) p[6];

    if (i==0)
    {
        switch (lp) {
            case 0:
            {
                /* p[6] never used here to move coefficients in the first time, */
                /* p[3] is used instead */
                p[6]->word[3] = 0x00001000;
                p[6]->word[4] = 0x00001000;
                p[6]->word[5] = 0x00000000;
                p[6]->word[7] = 0x00000000;
                p[6]->word[9] = 0x00000000;

                p[7]->word[3] = 0x00001000;
                p[7]->word[4] = 0x00001000;
                p[7]->word[6] = 0x00000000;
                p[7]->word[8] = 0x00000000;
                p[7]->word[10] = 0x00000000;
                break;
            }
            case 1:
            {
                p[6]->word[3] = 0x00ff0002;
                p[6]->word[4] = 0x00000c00;
                p[6]->word[5] = 0x00000005;
                p[6]->word[7] = 0x00000004;
                p[6]->word[9] = 0x00000800;

                p[7]->word[3] = 0x00000c00;
                p[7]->word[4] = 0x00ff0002;
                p[7]->word[6] = 0x00000005;
                p[7]->word[8] = 0x00000004;
                p[7]->word[10] = 0x00000800;
                break;
            }
            case 2:
            {
                p[6]->word[3] = 0x007f0002;
                p[6]->word[4] = 0x00000f00;
                p[6]->word[5] = 0x0000000e;
                p[6]->word[7] = 0x00000008;
                p[6]->word[9] = 0x00001000;

                p[7]->word[3] = 0x00000f00;
                p[7]->word[4] = 0x007f0002;
                p[7]->word[6] = 0x0000000e;
                p[7]->word[8] = 0x00000008;
                p[7]->word[10] = 0x00001000;
                break;
            }
            case 3:
            {
                p[6]->word[3] = 0x003f0002;
                p[6]->word[4] = 0x00000400;
                p[6]->word[5] = 0x00000007;
                p[6]->word[7] = 0x00000010;
                p[6]->word[9] = 0x00002000;
            }
        }
    }

```

```

    p[7]->word[3] = 0x00000400;
    p[7]->word[4] = 0x003f0002;
    p[7]->word[6] = 0x00000007;
    p[7]->word[8] = 0x00000010;
    p[7]->word[10] = 0x00002000;
    break;
}
case 4:
{
    p[6]->word[3] = 0x001f0002;
    p[6]->word[4] = 0x00000100;
    p[6]->word[5] = 0x00000003;
    p[6]->word[7] = 0x00000020;
    p[6]->word[9] = 0x00004000;

    p[7]->word[3] = 0x00000100;
    p[7]->word[4] = 0x001f0002;
    p[7]->word[6] = 0x00000003;
    p[7]->word[8] = 0x00000020;
    p[7]->word[10] = 0x00004000;
    break;
}
default:
    break;
}
}

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002001); /* send msg interrupt to PP0 */

p[6]->word[1] = p[6]->word[1] + jump_row[lp];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002002); /* send msg interrupt to PP1 */

p[6]->word[1] = p[6]->word[1] + jump_row[lp];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002004); /* send msg interrupt to PP2 */

p[6]->word[1] = p[6]->word[1] + jump_row[lp];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002008); /* send msg interrupt to PP3 */

```



```

p[6]->word[1] = p[6]->word[1] + jump_row[lp];

while((INTPEN & 0x10000)==0x00); /* poll PP0 */
INTPEN = 0x10000;                /* clear the interrupt flag */

*ptr = (long) p[7];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[7]->word[2] = p[7]->word[2] + jump_row[lp];

while((INTPEN & 0x20000)==0x00); /* poll PP1 */
INTPEN = 0x20000;                /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[7]->word[2] = p[7]->word[2] + jump_row[lp];

while((INTPEN & 0x40000)==0x00); /* poll PP2 */
INTPEN = 0x40000;                /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[7]->word[2] = p[7]->word[2] + jump_row[lp];

while((INTPEN & 0x80000)==0x00); /* poll PP3 */
INTPEN = 0x80000;                /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[7]->word[2] = p[7]->word[2] + jump_row[lp];

} /* end for i */

NOCACHE_INT(global_mean) = (NOCACHE_INT(mean_pp[0]) + NOCACHE_INT(mean_pp[1]) +
NOCACHE_INT(mean_pp[2]) + NOCACHE_INT(mean_pp[3])) / (512*240);

/* Do the appropriate number of columns depending on the scale */

if (lp!=4)
  for (i=0; i<vert_loop_index[lp]; i++)
  {
    if (i==0)
    {
      p[4]->word[1] = 0x80320000; /* Src address is DRAM */
      p[5]->word[2] = 0x80320000; /* Dst address is DRAM */
    }
  }

```

```

p[4]->word[2] = 0x00000000;          /* Dst address is internal */
*ptr = (long) p[4];

if (i==0)
{
    switch (lp) {
        case 0:
        {
            p[4]->word[3] = 0x00ef0010;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x00000000;
            p[4]->word[7] = 0x00000400;
            p[4]->word[9] = 0x00000000;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x00ef0010;
            p[5]->word[6] = 0x00000000;
            p[5]->word[8] = 0x00000400;
            p[5]->word[10] = 0x00000000;
            break;
        }
        case 1:
        {
            p[4]->word[3] = 0x000f0002;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x00000077;
            p[4]->word[7] = 0x00000004;
            p[4]->word[9] = 0x00000800;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x000f0002;
            p[5]->word[6] = 0x00000077;
            p[5]->word[8] = 0x00000004;
            p[5]->word[10] = 0x00000800;
            break;
        }
        case 2:
        {
            p[4]->word[3] = 0x001f0002;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x0000003b;
            p[4]->word[7] = 0x00000008;
            p[4]->word[9] = 0x00001000;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x001f0002;
            p[5]->word[6] = 0x0000003b;
            p[5]->word[8] = 0x00000008;
            p[5]->word[10] = 0x00001000;
            break;
        }
        case 3:
        {
            p[4]->word[3] = 0x000f0002;
            p[4]->word[4] = 0x000003c0;
            p[4]->word[5] = 0x0000001d;
            p[4]->word[7] = 0x00000010;
            p[4]->word[9] = 0x00002000;

            p[5]->word[3] = 0x000003c0;
            p[5]->word[4] = 0x000f0002;
            p[5]->word[6] = 0x0000001d;
            p[5]->word[8] = 0x00000010;
            p[5]->word[10] = 0x00002000;
            break;
        }
        default:

```

# NAWCWD TP 8442

```

        break;
    }
}

/* kick off TC to transfer columns of pixels from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002001);          /* send msg interrupt to PP0      */

p[4]->word[1] = p[4]->word[1] + jump_col[lp];          /* Src address is SDRAM */

/* kick off TC to transfer columns of pixels from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002002);          /* send msg interrupt to PP1      */

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM      */

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002004);          /* send msg interrupt to PP2      */

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM      */

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002008);          /* send msg interrupt to PP3      */

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM      */

p[5]->word[1] = 0x00000000;          /* Src address is internal      */
*ptr = (long) p[5];

while((INTPEN & 0x10000)==0x00); /* poll PP0 */
INTPEN = 0x10000;          /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */

while((INTPEN & 0x20000)==0x00); /* poll PP1 */

```

# NAWCWD TP 8442

```

INTPEN = 0x20000;          /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */

while((INTPEN & 0x40000)==0x00); /* poll PP2 */
INTPEN = 0x40000;          /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */

while((INTPEN & 0x80000)==0x00); /* poll PP3 */
INTPEN = 0x80000;          /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */

}

} /* end for lp */

#endif

*pp_stop_encode = 0;

#ifdef ENCODE_STREAM

/* Send over coefficients for maxval calculation */

p[6]->word[1] = 0x80320000; /* Src address is SDRAM */
p[6]->word[3] = 0x001f0002;
p[6]->word[4] = 0x00000100;
p[6]->word[5] = 0x00000003;
p[6]->word[7] = 0x00000020;
p[6]->word[9] = 0x00004000;

*ptr = (long) p[6];
p[6]->word[2] = 0x00000000; /* Dst address is internal */

/* kick off 128 word transfer from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

command(0x00002001); /* send msg interrupt to PP0 */

p[6]->word[1] = p[6]->word[1] + jump_row[4];

/* kick off 128 word transfer from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

```

# NAWCWD TP 8442

```

command(0x00002002);          /* send msg interrupt to PP1      */

p[6]->word[1] = p[6]->word[1] + jump_row[4];

/* kick off 128 word transfer from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

command(0x00002004);          /* send msg interrupt to PP2      */

p[6]->word[1] = p[6]->word[1] + jump_row[4];

/* kick off 128 word transfer from SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

command(0x00002008);          /* send msg interrupt to PP3      */

while((INTPEN & 0x0F0000)!=0x0F0000); /* wait for PP0,1,2,3 */
INTPEN = 0x0F0000; /* clear the interrupt flag */

temp_maxval = NOCACHE_INT(local_maxval[0]);

if (NOCACHE_INT(local_maxval[1]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[1]);
if (NOCACHE_INT(local_maxval[2]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[2]);
if (NOCACHE_INT(local_maxval[3]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[3]);

i=0;
while(temp_maxval!=1)
{
    i++;
    temp_maxval = temp_maxval>>1;
}
temp_maxval = 1<<i;

NOCACHE_INT(global_maxval) = i;

command(0x0000200F);          /* send msg interrupt to PP0,1,2,3 */

/*****

#ifdef HOST
/* Tell host where bit stream can be found */

RetVal = CilWriteMailbox(0,stream_addr[flip]);

if (RetVal != CIL_OK)
{
    /*** Cannot write to mailbox ***/
    while(1);
    return;
}

#endif

table_pointer = stream_addr[flip] + 8;

ppnexttask[0] = 0;
ppnexttask[1] = 0;
ppnexttask[2] = 0;
ppnexttask[3] = 0;

current_block = 0;
ppdone = 0;

while (ppdone!=4)
{
    if ((INTPEN & 0x0F0000)!=0x00)

```

```

{
    DONE1 = 0;

    /* find out which PP requested service */

    for (pprequesting=0;pprequesting<4;pprequesting++)
    {
        if ((INTPEN & (1<<(16+pprequesting))) != 0) break;
    }

    /* clear the interrupt flag */

    INTPEN = 0x10000<<pprequesting;

    /* Move bitstream from internal to SDRAM, and move first block pair of coefficients
    if necessary */

    if ((ppnexttask[pprequesting] == 3) && (!DONE1))
    {
        p[9]->word[1] = 0x01000630 + (pprequesting<<12);      /* Src address is internal
        Parameter RAM */
        p[9]->word[2] = table_address[pprequesting];          /* Dst address is external
        SDRAM */

        *ptr = (long) p[9];

        /* Read the number of bytes that need to be transferred */

        p[9]->word[3] = p[9]->word[4] = p[9]->word[10] = table_size[pprequesting];

        /* kick off transfer from internal to SDRAM */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        if (current_block != 40)
        {
            p[1]->word[1] = 0x80320000 + (current_block>>3)*0x04000 + (current_block&7)*64;
            /* Src for block changes */
            p[1]->word[2] = 0x00008000 + (pprequesting<<12);      /* Dst address is
            internal RAM2 */

            *ptr = (long) p[1];

            /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
            internal */
            PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

            p[1]->word[1] = p[1]->word[1] + 0x200;                /* Src for block changes */
            p[1]->word[2] = p[1]->word[2] + 0x400;                /* Dst address is internal RAM2
            */

            /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
            internal */
            PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

            /* write out TBYTES value for this block to the appropriate PP */

            table_size[pprequesting] = local_alloc[current_block];

            *(unsigned short *) (0x01000600 + (pprequesting<<12)) =
            table_size[pprequesting];

            command(0x00002000 + (1<<pprequesting));          /* send msg interrupt to PP
            that requested service */

            table_address[pprequesting] = table_pointer;
            table_pointer = table_pointer + table_size[pprequesting];
        }
    }
}

```

```

        ppinfo[pprequesting] = current_block;
        ppnexttask[pprequesting] = 1;
        current_block = current_block + 1;
    }
    else ppdone = ppdone + 1;

    DONE1 = 1;

}

/* Give PP second and third block pairs to process */

if ((ppnexttask[pprequesting] > 0) && (ppnexttask[pprequesting] < 3)) && (!DONE1))
{
    p[1]->word[1] = 0x80320000 + (ppinfo[pprequesting]>>3)*0x04000 +
(ppinfo[pprequesting]&7)*64 + 0x14000*ppnexttask[pprequesting];    /* Src for block
changes */
    p[1]->word[2] = 0x00008000 + (pprequesting<<12);    /* Dst address is
internal RAM2 */

    *ptr = (long) p[1];

    /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    p[1]->word[1] = p[1]->word[1] + 0x200;    /* Src for block changes */
    p[1]->word[2] = p[1]->word[2] + 0x400;    /* Dst address is internal RAM2
*/

    /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    command(0x00002000 + (1<<pprequesting));    /* send msg interrupt to PP
that requested service */

    ppnexttask[pprequesting] += 1;

    DONE1 = 1;

}

/* Give PP first block pair to process */

if ((ppnexttask[pprequesting] == 0) && (!DONE1))
{
    p[1]->word[1] = 0x80320000 + (current_block>>3)*0x04000 + (current_block&7)*64;
/* Src for block changes */
    p[1]->word[2] = 0x00008000 + (pprequesting<<12);    /* Dst address is
internal RAM2 */

    *ptr = (long) p[1];

    /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    p[1]->word[1] = p[1]->word[1] + 0x200;    /* Src for block changes */
    p[1]->word[2] = p[1]->word[2] + 0x400;    /* Dst address is internal RAM2
*/

    /* kick off 1024 byte (32x16 words) coefficient block transfer from DRAM to
internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    /* write out TBYTES value for this block to the appropriate PP */

```

```

    table_size[pprequesting] = local_alloc[current_block];

    *(unsigned short *) (0x01000600 + (pprequesting << 12)) = table_size[pprequesting];

    command(0x00002000 + (1 << pprequesting));          /* send msg interrupt to PP
that requested service */

    table_address[pprequesting] = table_pointer;
    table_pointer = table_pointer + table_size[pprequesting];

    ppinfo[pprequesting] = current_block;
    ppnexttask[pprequesting] = 1;
    current_block = current_block + 1;

    DONE1 = 1;
}

}

}

*pp_stop_encode = 1;
command(0x0000200F);          /* send msg interrupt to all PPs */

#endif

/* Write dynamic values to the bitstream */

temp_ulong = 0;
temp_ulong = temp_ulong | (tbuf_pp0[0] << 24);
temp_ulong = temp_ulong | (global_mean << 16);
temp_ulong = temp_ulong | (global_maxval << 8);

NOCACHE_INT(* (UINT32 *) stream_addr[flip]) = temp_ulong;

temp_ulong = *compression_ratio;

#ifdef variable_compression
    NOCACHE_INT(* (UINT32 *) (stream_addr[flip] + 4)) = temp_ulong & 0x000000ff;
#else
    NOCACHE_INT(* (UINT32 *) (stream_addr[flip] + 4)) = temp_ulong;
#endif

/* Flip which buffer we are writing to */

flip = 1 - flip;

/* Tell Host it can now get the latest bitstream */

#ifdef HOST
    ReturnVal = CilRaiseSignalNumber(FrameDoneBit);
    if (ReturnVal != CIL_OK)
    {
        /*** Cannot raise signal ***/
        return;
    }
#endif

/* Calculate and display latest frame timings */

proc_time = 0xffffffff - TCOUNT;

#ifdef SHOWDISPLAY

if (Semaphore == 0)
{

```



```

p[7]->word[3] = 0x00001000;
p[7]->word[4] = 0x00001000;
p[7]->word[6] = 0x00000000;
p[7]->word[8] = 0x00000000;
p[7]->word[10] = 0x00001000;

/* DRAM (8 bit data) -> VRAM (raw image) */

p[2]->link = p[2]; /* point to this PT */
p[2]->word[0] = 0x80000000; /* linear to VRAM */
p[2]->word[1] = 0x80300000; /* Src address is SDRAM */
p[2]->word[2] = 0xb4000000; /* Dst address is VRAM */
p[2]->word[3] = 0x00110010; /* Src B count Src A count */
p[2]->word[4] = 0x00110010; /* Dst B count Dst A count */
p[2]->word[5] = 0x00; /* Src C count */
p[2]->word[6] = 0; /* Dst C count */
p[2]->word[7] = 0xb0; /* Src B pitch */
p[2]->word[8] = 0x400; /* Dst B pitch */
p[2]->word[9] = 0x00; /* Src C pitch */
p[2]->word[10] = 0x0000; /* Dst C pitch */
p[2]->word[11] = 0; /* Src transparency upper */
p[2]->word[12] = 0; /* Src transparency lower */
p[2]->word[13] = 0; /* Reserved */
p[2]->word[14] = 0; /* Reserved */

temp_time = 1.0/(encode_time*0.000000025);

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
    else
    {
        temp_str = 160;
    }

    p[2]->word[1] = (long)&number_pixels;
    p[2]->word[1] += temp_str;
    p[2]->word[2] = 0xb4080000 + i*16;

    *ptr = (long) p[2];

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from SDRAM to VRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);
}

temp_time = proc_time*0.000025;

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
    else

```

```

    {
        temp_str = 160;
    }

    p[2]->word[1] = (long)&number_pixels;
    p[2]->word[1] += temp_str;
    p[2]->word[2] = 0xb4084800 + i*16;

    *ptr = (long) p[2];

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from SDRAM to VRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);
}

/* Restore original p[2] PT array */
/* SDRAM (8 bit data) -> VRAM (raw image) */

    p[2]->link = p[2]; /* point to next PT */
    p[2]->word[0] = 0x80000000; /* linear to VRAM */
    p[2]->word[1] = 0x80300000; /* Src address is DRAM */
    p[2]->word[2] = 0xb4000000; /* Dst address is VRAM */
    p[2]->word[3] = 0x00038000; /* Src B count Src A count */
    p[2]->word[4] = 0x00ff0200; /* Dst B count Dst A count */
    p[2]->word[5] = 0x00; /* Src C count */
    p[2]->word[6] = 0; /* Dst C count */
    p[2]->word[7] = 0x8000; /* Src B pitch */
    p[2]->word[8] = 0x800; /* Dst B pitch */
    p[2]->word[9] = 0x00; /* Src C pitch */
    p[2]->word[10] = 0x0000; /* Dst C pitch */
    p[2]->word[11] = 0; /* Src transparency upper */
    p[2]->word[12] = 0; /* Src transparency lower */
    p[2]->word[13] = 0; /* Reserved */
    p[2]->word[14] = 0; /* Reserved */

}

#endif

} /* end while */

} /* end task */

extern int ep_runpp0;

main()
{
    int i;
    unsigned int temp;
    unsigned int *src_ptr = (unsigned int *)0x90080000;
    unsigned int *dst_ptr = (unsigned int *)0x80000000;

    /* REFCNTL = 0xffff0138; */ /* setup up dram and sdram to correct refresh rate for 40 Mhz C80*/
    REFCNTL = 0xffff0186; /* setup up dram and sdram to correct refresh rate for 50 Mhz C80*/

    command(0xc000000f); /* reset and halt PP0,1,2,3 */

    *(int *)0x010001b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010011b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010021b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010031b8 = (int)&ep_runpp0; /* initialize task vector */

    /* upload PP code */

    for (i=0;i<34000;i++)

```

```

    {
        temp = *src_ptr++;
        *dst_ptr++ = temp;
    }

for (i=0;i<20000;i++)
{
    temp = temp + 1;
    temp = temp -1;
}

command(0x3000000f);          /* start PP0,1,2,3 by unhalting it */
                              /* all will take its task interrupt*/

/* Basic init functions */

#ifdef HOST
    InterruptInit();          /* Init ME interrupts */
#endif

/* Basic init functions */

PtReqInit();                  /* Init the ME PT functions */
TaskInitTasking();            /* Init tasking */
IclInstallPtdMalloc();         /* Install protected malloc and free function to ME */
IclPTInit(15);                /* Init the Icl PT server task with a priority of 15 */

#ifdef HOST
/*
** Initialise the Cil
** Declare 4 buffers of 256 bytes each.
** These buffers are not used here - choose minimum sizes.
*/
CilInit(4,256);
#endif

TaskResume(TaskCreate(-1,task, NULL, 14, 4096)); /* Start task */

while(l==1); /* loop */
}

/*****
*
*   Function : SignalHandler
*   Args      : UINT32 Signals
*
*   Description:
*       Signals          Signals raised by host
*
*       SignalHandler will be called when host raises signal
*
*   Return Values:
*       None
*
*****/

void
SignalHandler(UINT32 Signals)
{
    if ((Signals & HostRequestBitMask)!=0)
    {
        /*
        ** Host has requested a block of data
        */
        /* TaskSignalSema(Semaphore); */
        Semaphore = 1;
    }
    /*
    ** Ignore any other signals raised - they're not for us

```

```

    */
}

/* Initialize the adaptive bit allocation tables on the PPs */

/*****
 *
 * Function : init_alloc_table
 * Args    : UINT32 index
 *
 * Description:
 *     Signals      index into the compression table
 *
 *     init_alloc_table will be called when host changes the
 *     compression ratio
 *
 * Return Values:
 *     None
 *
 *****/

void
init_alloc_table(UINT32 index)
{
    int t_alloc;
    int p_alloc;
    int r_alloc;
    int k,lp;

    t_alloc = YSIZE*FSIZE/comp_table[index] - headersize;
    p_alloc = t_alloc/((AY*AX)/6);
    r_alloc = t_alloc%((AY*AX)/6);

    for(k=0;k<((AY*AX)/6);k++)
        local_alloc[k] = p_alloc;

    lp = 0;
    while (r_alloc>0)
    {
        local_alloc[lp++] ++;
        r_alloc--;
    }
}

```

# NAWCWD TP 8442

```

*****/
*
*   Filename : pcic80.cmd
*
*   Description:
*
*   PCI/C80 linker file for the MP program
*
*****/
-C
-heap 0x100000
-stack 0x10000
-l mp_cio.lib
-l \pcic80\lib\mp_task.lib
-l mp_int.lib
-l mp_rts.lib
-l mp_ptreq.lib
-l mp_ppcmd.lib
-l ppcmd.lib
-l \pcic80\lib\icl.lib
-l \pcic80\lib\vol.lib
-l \pcic80\lib\bgl.lib
-l \pcic80\lib\vil24.lib
-l \pcic80\lib\cil.lib

MEMORY
{
    RAM0 : o=0x00000000  l = 0x00800
    RAM1 : o=0x00000800  l = 0x00800
    RAM2 : o=0x00008000  l = 0x00800
    RESERV : o=0x01000000  l = 0x00200
    MPFRAM1 : o=0x01010580  l = 0x00280
    MPFRAM : o=0x010007D8  l = 0x00028
    SDRAM : o=0x80000000  l = 0x300000
    DRAM : o=0x90000000  l = 0x80000
    DRAM2 : o=0x90080000  l = 0x100000
    UNINIT : o=0x90180000  l = 0x180000
    IMAGE : o=0x80300000  l = 0x80000
    SPOT : o=0x80380000  l = 0x10000
    VRAM_PAL : o=0xB0000000  l = 0x200000
    VRAM_VGA : o=0xB4000000  l = 0x400000
}

SECTIONS
{
    /*
    * The following section must be defined for all programs that
    * use the CIL. The section must appear in the first 8Mb
    * of DRAM and must be long enough to include all buffers
    * plus 128 bytes. This example is big enough for 4*256byte
    * buffers.
    *
    * See the user guide for more information.
    */
    .lsidram : {
        _CilDRAMBase = .;
        . += 0x600;
    } > DRAM

    .text : > DRAM
    .cinit : > DRAM
    .const : > DRAM
    .switch : > DRAM
    .data : > DRAM
    .bss : > DRAM
    .cio : > DRAM
    .pcinit : > DRAM

    .ptext : load > DRAM2, run SDRAM
    font : load > DRAM2, run SDRAM

```

NAWCWD TP 8442

```
.system      :    > UNINIT
.stack       :    > UNINIT

sh_vars      :    > MPFRAM
mp_vars      :    > MPFRAM1

rawimage     :    > IMAGE
stream       :    > SPOT
)
```

# NAWCWD TP 8442

```

/*****
*
*   Filename : pcic80a.cmd
*
*   Description:
*
*   PCI/C80 linker file for the PP program
*
*****/
-pc
-l d:\mvp\src\newlib\pp_rts.lib /* New Library w/o ATEXTIT extra variables */

-pstack 256

MEMORY
{
    RAM0 : o=0x00000000 l = 0x00800
    RAM1 : o=0x00000800 l = 0x00800
    RAM2 : o=0x00008000 l = 0x00800
    RESERV : o=0x01000000 l = 0x00200
    PRAM : o=0x01000200 l = 0x00600
    SDRAM : o=0x80000000 l = 0x800000
    DRAM : o=0x90400000 l = 0x400000
    VRAM_PAL : o=0xB0000000 l = 0x200000
    VRAM_VGA : o=0xB4000000 l = 0x400000
}

SECTIONS
{
    .text : > DRAM
    .ptext : > DRAM
    .cinit : > DRAM
    .const : > DRAM
    .switch : > DRAM
    .data : > DRAM
    .bss : > DRAM
    .cio : > DRAM
    .sysmem : > DRAM
    .stack : > DRAM

    .pcinit : > DRAM

    .pbss : (PASS) > PRAM
    .psysmem: (PASS) > PRAM
    .pstack : (PASS) > PRAM
}

```

```

/*****
**
** subpass.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutine:
**
** $subpass
**
*****/

.global $subpass

.global $stats_flag      ;unsigned char pointer *(xba + $stats_flag)
.global $char_to_index   ;unsigned char pointer &*(xba + $char_to_index)
.global $stats_val        ;signed short pointer *(xba + $stats_val)
.global $BITE             ;signed int sw *(xba + $BITE)
.global $STOP             ;signed int sw *(xba + $STOP)

.global $update_model
.global $encode_symbol

.global $sym_index
.global $sym_array
.global $do_syms

.global $list
.global $list_index

tempd1 .set d4
tempd2 .set d2
tempd3 .set d5
tempd4 .set d1

m .set d6
tt .set d7

list .set a12
stats_val .set a4

i .set 20
j .set 24
t .set 28

.align 512

/*****
*
* Function : $subpass
* Args : t
* Passed in : d1
*
* Description:
* t - the current THRESH >> BITE value.
*
* $subpass will be called to perform a subordinate pass over
* the coefficients.
*
* Return Values:
* none
*
*****/

$subpass:
d0 = &(sp --= 32)
*(sp + 12) =w iprs

*(sp + 8) =w d6

```



```

||      *(sp + 16) =w  a4

      *(sp + 4) =w  a12
||      *(sp + 0) =w  d7

      *(sp + t) = d1

      tempd1 =sw  *(xba + $STOP)
      tempd1 = tempd1 - 0
      br =[ne] done

      tempd1 =uh  *(xba + $list_index)
      tempd1 = tempd1 - 0
      br =[eq] done
      nop
      nop

mloop:
      list =uw  *(xba + $list)
      stats_val =uw  *(xba + $stats_val)
      tempd1 =uh  *(xba + $list_index)
      *(sp + i) = zero
                                     ;repeat loop list_index times
                                     ;i = 0

      le0 = firstloop
      lrs0 = tempd1 - 1
      nop
      x8 = *(sp + i)

      tempd1 = x8 + 1
      m =uh  *(list + [x8])
||      *(sp + i) = tempd1

      x0 = m
      tempd1 =uw  *(xba + $BITE)
      le1 = endjloop
      lrs1 = tempd1 - 1

      tt = *(sp + t)
      tt = tt >>u 1

jloop:
      a0 = &*(pba + $sym_array)

      tempd2 =sh  *(stats_val + [x0])
||      tempd1 = tt
      tempd2 = tempd2 - 0
      tempd4 = [.nvz] 1 || tempd4 = [le.nvz] zero

      tempd1 = [le] -tt

      tempd2 = tempd2 - tempd1
      *(stats_val + [x0]) =sh tempd2

      x0 =uh  *(pba + $sym_index)
      tempd2 = x0 + 1

; if (sym_index>240) do_syms();

      tempd1 = tempd2 - 240

      call1 =[gt] $do_syms
      *(pba + $sym_index) =uh tempd2

;sym_array[sym_index++] = sym;

      *(a0 + [x0]) =ub tempd4
||      tt = tt >>u 1

endjloop:

```

# NAWCWD TP 8442

```
x0 = m  
  
firstloop:  
    x8 = *(sp + i)  
  
done:  
    a12 =sw *(sp + 4)  
    a4  =sw *(sp + 16)  
    br  = *(sp + 12)  
  
    d6 =sw *(sp + 8)  
    || d7 =sw *(sp + 0)  
    d0 = &*(sp += 32)
```

# NAWCWD TP 8442

```

/*****
**
** ztr.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $comp_ztr
**
*****/

        .global $comp_ztr

/*****
*
* Function : $comp_ztr
* Args    : s
* Passed in : d1
*
* Description:
*   s - the subblock to process
*
*   $comp_ztr will be called to calculate the ztl array for a
*   single subblock.
*
* Return Values:
*   None
*
*****/

$comp_ztr:
        lctl = 0x0 ;reset looping capability

        d2 = d1 << 9
        x0 = d2 + 0x1fe
        nop
        a0 = &*(dba + x0) ;stats_val[s][m=255] = stats_val + (s*256 + m) *(2
bytes/word) ;(0x1fe, 0x3fe, 0x5fe, 0x7fe, 0x9fe, 0xbfe)

        d2 = d1 << 7
        x0 = d2 + 0xc7e
        nop
        a2 = &*(dba + x0) ;ztl[s][p=63] = ztl + (s*64 + p) *(2 bytes/word)
        nop ;(0xc7e, 0xcfe, 0xd7e, 0xdfe, 0xe7e, 0xefe)
        a8 = a2

;clear out ztl[s][0..63] array
        x0 = d2 + 0xc00
        nop
        a1 = &*(dba + x0)
        lrse0 = 31
        d1 = 0
        nop

        *(a1++=[1])= d1

        lr0 = 3 ;first innerloop
        lr1 = 3 ;second innerloop
        lr2 = 11 ;outerloop

        le0 = firstloopend2
        ls0 = firstloopstart2
        le1 = secondloopend2
        ls1 = secondloopstart2
        le2 = outerloopend2
        ls2 = outerloopstart2

```

# NAWCWD TP 8442

```

nop
lctl1 = 0xba9 ;associate le0 with lc0, le1 with lc1, and le2 with lc2

nop
nop

d4 =sh *(a0--[1])

outerloopstart2:
    nop

firstloopstart2:

    d4 = |d4|
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    || d3 =sh *(a0--[1])

    d3 = |d3|
    d3 = 31 - lmo(d3)
    d3 = 1<<d3

    d2 = d2 | d4 | d3
    || d4 =sh *(a0--[1])

firstloopend2:
    *(a2--[1]) =h d2

    a7 =uh *(a2+=[4])

secondloopstart2:

    d4 = |d4|
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    || d3 =sh *(a0--[1])

    d3 = |d3|
    d3 = 31 - lmo(d3)
    d3 = 1<<d3

    d2 = d2 | d4 | d3
    || d4 =sh *(a0--[1])

secondloopend2:
    *(a2--[1]) =h d2

outerloopend2:
    nop

    lr0 = 1      ;first innerloop
    lr1 = 1      ;second innerloop
    lr2 = 5      ;outerloop

    le0 = firstloopend3
    ls0 = firstloopstart3
    le1 = secondloopend3
    ls1 = secondloopstart3
    le2 = outerloopend3
    ls2 = outerloopstart3

    nop
    nop

outerloopstart3:
    nop

```

firstloopstart3:

```

    d4 = |d4|
|| d3 =h *(a8)
    d4 = 31 - lmo(d4)
|| d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

    d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

```

    d4 = |d4|
|| *(a2) =h d2
    d4 = 31 - lmo(d4)
|| d3 =h *(a8)
    d4 = 1<<d4
|| d2 =h *(a2)
    d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

    d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

firstloopend3:

```

    *(a2--=[1]) =h d2

    a7 =uh *(a2++=[2])

```

secondloopstart3:

```

    d4 = |d4|
|| d3 =h *(a8)
    d4 = 31 - lmo(d4)
|| d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

    d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

```

    d4 = |d4|
|| *(a2) =h d2
    d4 = 31 - lmo(d4)
|| d3 =h *(a8)
    d4 = 1<<d4
|| d2 =h *(a2)
    d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

    d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

secondloopend3:

```

    *(a2--=[1]) =h d2

```

outerloopend3:

```

    nop

```

```

    lr0 = 2      ;first loop

```

```

    le0 = firstloopend4
    ls0 = firstloopstart4

```

```

    nop
    nop

```

firstloopstart4:

```

    d4 = |d4|
|| d3 =h *(a8)
    d4 = 31 - lmo(d4)
|| d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
|| d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
|| *(a8--[1]) =h d3

```

```

    d4 = |d4|
|| *(a2) =h d2
    d4 = 31 - lmo(d4)
|| d3 =h *(a8)
    d4 = 1<<d4
|| d2 =h *(a2)
    d3 = d3 | d4
|| d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
|| *(a8--[1]) =h d3

```

```

    *(a2) =h d2

```

```

    d4 = |d4|
|| d3 =h *(a8)
    d4 = 31 - lmo(d4)
|| d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
|| d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
|| *(a8--[1]) =h d3

```

```

    d4 = |d4|
|| *(a2) =h d2
    d4 = 31 - lmo(d4)
|| d3 =h *(a8)
    d4 = 1<<d4
|| d2 =h *(a2)
    d3 = d3 | d4
|| d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
|| *(a8--[1]) =h d3

```

```

    *(a2--[1]) =h d2

```

firstloopend4:

```

    nop

```

```

    d4 = |d4|
|| d3 =h *(a8)
    d4 = 31 - lmo(d4)
|| d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
|| d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
|| *(a8--[1]) =h d3

```

```

    d4 = |d4|

```

# NAWCWD TP 8442

```

|| *(a2) =h d2
  d4 = 31 - lmo(d4)
|| d3 =h *(a8)
  d4 = 1<<d4
|| d2 =h *(a2)
  d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

  d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

```

  *(a2) =h d2

```

```

  d4 = |d4|
|| d3 =h *(a8)
  d4 = 31 - lmo(d4)
|| d2 =h *(a2)
  d4 = 1<<d4
  d3 = d3 | d4
|| d4 =sh *(a0--=[1])

```

```

  d2 = d2 | d3
|| *(a8--=[1]) =h d3

```

```

  d4 = |d4|
|| *(a2) =h d2
  d4 = 31 - lmo(d4)
|| d3 =h *(a8)
  d4 = 1<<d4
|| d2 =h *(a2)
  d3 = d3 | d4

```

```

  d2 = d2 | d3
|| *(a8) =h d3

```

```

  *(a2) =h d2

```

```

br = iprs
nop
nop

```

## INTRAFRAME DECODING

```

;Written by Jim Witham
; 3-27-97
; addconv.s
; To replace the following C Code that just took too much time...
    for(i=0;i<2048;i++)
;       {
;           img[i] = img[i] + oldmean
;
;           pic[i] = img[i];
;           if (img[i]>255)
;               pic[i] = 255;
;           if (img[i]<0)
;               pic[i] = 0;
;       }
;

.global $add_n_conv8

$add_n_conv8:
    d1 = d1<<2
    d6 =rh d1                ;d1 is mean that is passed in
                                ;replicate it to both words

    sr = 0xad
    x0 = 1
    x1 = 2

    d0 = 0
    d4 = 0x04000400          ; 0400>>2 = 0100
    d5 = 0x03fc03fc          ; 03fc>>2 = 00ff
    a2 = &*(dba + 1)

    le2 = L48
    lrs2 = 1023

    a1 = &*(dba)
    a0 = &*(dba)

    d1 = *(a1++=[x0])
    d1 =me d1 + d6
    d3 = (d0 & @mf) | (d1 & ~@mf) ;if (val+mean<0) then d3 = 0
else d3 = d1
    d2 =me d1 - d4
    d3 = (d3 & @mf) | (d5 & ~@mf) ;if (val+mean>0x400) then d3 =
0x3fc else d3 = d3

    d3 = d3 >>u 2            ; shift down to proper scaling
    *(a2++=[x1]) =b d3
    || d2 =h1 d3

```



NAWCWD TP 8442

L48:

\*(a0++=[x1]) =b d2

sr = 0x36

br = iprs

nop

nop

## NAWCWD TP 8442

```
/* DECLARATIONS FOR ARITHMETIC CODING AND DECODING */

/* Size of arithmetic code values */

#define Code_value_bits 9 /* Number of bits in a code value */
typedef short code_value; /* Type of arithmetic code value */

#define Top_value (((long)1<<Code_value_bits)-1) /* Largest code val
*/

/* Half and Quarter points in code value range */

#define First_qtr (Top_value/4+1) /* Points after first quarter */
#define Half (2*First_qtr) /* Points after first half */
#define Third_qtr (3*First_qtr) /* Points after third quarter */
```

## NAWCWD TP 8442

```
mpcl -s -g -c -i\pcic80\include mp.c  
mvplnk -x mp.obj num2.obj \nawcwip.obj \dsp_util.obj \frame_ra.obj  
pp0.out pcic80.cmd -o mp.out -m mp.map
```

# NAWCWD TP 8442

```
erase *.o
ppcl -s -k main.c
ppcl -s -k j_dec.c
ppcl -s -k modlp.c
ppcl -o2 form.c
ppasm decbits.s
ppasm hvdec.s
ppasm addconv.s
ppasm quick.s
ppasm newardec.s
mvplnk -x main.o hvdec.o decbits.o modlp.o addconv.o j_dec.o newardec.o
quick.o form.o pcic80a.cmd -t runpp0 -o pp0.out -m pp0.map
```

# NAWCWD TP 8442

```

/*****
**
** decbits.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $new_dec_dbits_ri
** $new_dec_dbits_nri
** $fast_cache_syms
** $new_decode_symbol
** I_DIV_JW2
** $new_update_model
** $new_dec_dbits3_ri
** $new_input_bit
** $quick_clear
**
*****/

.global $new_dec_dbits_ri
.global $new_dec_dbits_nri

.global $THRESH          ;signed   int    sw *(xba + $THRESH)
.global $stats_flag      ;unsigned char pointer *(xba + $stats_flag)

.global $stats_apx
.global $index_to_char

.global $freq
.global $No_of_symbols
.global $char_to_index

.global $TMASK           ;unsigned short uh *(xba + $TMASK)
.global $BITE            ;signed   int    sw *(xba + $BITE)

.global $T_BYTES
.global $byte_stream
.global $STOP

.global $bit_index

.global $update_model2

.global $list
.global $list_index

; needed for cacheing of symbols

.global $cache_syms
.global $symbol_index
.global $save_stop

.global $pruned_children

CACHE      .set 1
FRED2      .set 1

```

```

;*****
;* FUNCTION DEF: $new_dec_dbits_ri( m,c1,c3, s)
;*
;*
;*****

tempd1      .setd1
tempd2      .setd2
tempd3      .setd3
tflg        .setd4
tmask       .setd5
symbol      .setd6
t           .setd7

        .align 512

$new_dec_dbits_ri:

        d0 = &(sp --= 40)
        *(sp + 16) =w  iprs

        *(sp + 12) =w  a4
||      *(sp + 8) =w  d6

        *(sp + 4) =w  a12
||      *(sp + 0) =w  d7

;compute sym

        *(sp + 24) = d1
        *(sp + 28) = d2
        *(sp + 32) = d3
        *(sp + 36) = d4

        .if CACHE == 0

        call = $new_decode_symbol
        nop
        nop

        x8 = d5
        a12 = &(xba + $index_to_char)
        nop
        symbol =ub *(a12 + x8) ;d1 = index_to_char[sym]

        call = $new_update_model
        nop
        d1 = d5

        .else

; symbol = save_symbol[symbol_index]
; STOP = save_stop[symbol_index]
; symbol_index = symbol_index + 1;
; if (symbol_index==16) cache_syms();

        x8 =ub *(xba + $symbol_index) ;get next symbol
        a12 =uw *(xba + $save_symbol)
        a11 = &(xba + $save_stop) ;get next STOP value
        symbol =ub *(a12 + x8)

```

```

d1 =ub *(a11 + x8)
*(xba + $STOP) =uw d1

x8 = x8 + 1
*(xba + $symbol_index) =ub x8
x8 = x8 - 16 ;call cache_syms if we need more symbols
; call = [eq] $cache_syms
call = [eq] $fast_cache_syms
nop
nop

.endif

d1 = *(sp + 24)
d2 = *(sp + 28)
d3 = *(sp + 32)
d4 = *(sp + 36)

a1 =uw *(xba + $stats_apx) ;*(a1) = stats_apx[s*256]
a2 =uw *(xba + $stats_flag)

d5 = d4 << 9
a1 = a1 + d5

d5 = d4 << 8
a2 = a2 + d5

d5 = d1
d5 = d5 + (d4 << 8)

*(sp + 20) =uh d5

a0 = a2 ;*(a0) = stats_flag[s*256]

x0 = d1 ;x0 = m
x1 = d3 ;*(a2 + [x1]) = stats_flag[c3]
x2 = d3 + 1 ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

a2 = a2 + d2 ;*(a2) = stats_flag[s*256 + c1]
;*(a2 + [1]) = stats_flag[c2 = c1 + 1]

;*(a1 + [x0]) = stats_apx[s*256 + m]
;*(a0 + [x0]) = stats_flag[s*256 + m]
;*(a2) = stats_flag[s*256 + c1]
;*(a2 + [1]) = stats_flag[s*256 + c2] (c2 = c1 + 1)
;*(a2 + [x1]) = stats_flag[s*256 + c3]
;*(a2 + [x2]) = stats_flag[s*256 + c4] (c4 = c3 + 1)

tempd1 =sw *(xba + $BITE)
tempd1 = 1 << tempd1

t = symbol - tempd1
br = [le] sym_gt1
; nop
nop

tempd1 = 3
*(a0 + [x0]) =ub tempd1

x8 =uh *(pba + $list_index)
a15 = x8 - 254
br = [ge] nosig

```

```

    tempd1 =uh *(sp + 20)
    a12 =uw *(pba + $list)
    tempd2 = x8 + 1
    *(a12 + [x8]) =uh tempd1
    *(pba + $list_index) =uh tempd2

nosig:

; stats_apx[m] = ((t*THRESH)>>(BITE-1)) + (THRESH>>BITE);

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

    tflg = 1 - tempd1

    tmask =u tempd2 * t
    tempd1 = - tempd1

    tempd3 = (tmask >>u -tflg)

    br = done
    tempd3 = tempd3 + (tempd2 >>u -tempd1)
    *(a1 + [x0]) =h tempd3

sym_gt1:
    t = symbol - 1
    br =[le] sym_eq0
;    nop
    nop

    tempd1 = 1
    *(a0 + [x0])=ub tempd1

    x8 =uh *(pba + $list_index)
    a15 = x8 - 254
    br =[ge] nosig2
    tempd1 =uh *(sp + 20)
    a12 =uw *(pba + $list)
    tempd2 = x8 + 1
    *(a12 + [x8]) =uh tempd1
    *(pba + $list_index) =uh tempd2

nosig2:

; stats_apx[m] = ((t*THRESH)>>(BITE-1)) + (THRESH>>BITE);

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

    tflg = 1 - tempd1

    tmask =u tempd2 * t
    tempd1 = - tempd1

    tempd3 = -(tmask >>u -tflg)

    br = done
    tempd3 = tempd3 - (tempd2 >>u -tempd1)
    *(a1 + [x0]) =h tempd3

```



```

sym_eq0:
    t = symbol - 0
    br = [ne] done
;    nop
;    nop

    tempd2 =ub *(a2)
    tempd2 = tempd2 | 4
    *(a2) =ub tempd2

    tempd2 =ub *(a2 + [1])
    tempd2 = tempd2 | 4
    *(a2 + [1]) =ub tempd2

    tempd2 =ub *(a2 + [x1])
    tempd2 = tempd2 | 4
    *(a2 + [x1]) =ub tempd2

    tempd2 =ub *(a2 + [x2])

    tempd2 = tempd2 | 4
    *(a2 + [x2]) =ub tempd2

    a0 = &*(pba + $pruned_children)
    x0 = *(sp + 36)
    nop
    tempd1 =ub *(a0 + x0)
    tempd1 = tempd1 + 1
    *(a0 + x0) =ub tempd1

done:
    a12 =w *(sp + 4)
    a4  =w *(sp + 12)

    br = *(sp + 16)

    d6 =sw *(sp + 8)
    || d7 =sw *(sp + 0)
    d0 = &*(sp += 40)

;    branch occurs here

;*****
; * FUNCTION DEF: $new_dec_dbits_nri( m,c1,c3, s) *
; *                               d1,d2,d3,d4 *
;*****

$new_dec_dbits_nri:

    a2 =uw *(xba + $stats_flag)
    x8 = d4
    d4 = d4 << 8
    a2 = a2 + d4

    a0 = a2                ;*(a0 + [x0])          = stats_flag[s*256 + m]

    x0 = d1                ;x0 = m
    x1 = d3                ;*(a2 + [x1]) = stats_flag[s*256 + c3]
    x2 = d3 + 1            ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

    a2 = a2 + d2           ;*(a2)              = stats_flag[s*256 + c1]
                           ;*(a2 + [1])      = stats_flag[c2 = c1 + 1]

```

```

tempd2 =ub *(a0 + [x0])

tempd2 = tempd2 & 251
*(a0 + [x0]) =ub tempd2

tempd2 =ub *(a2)
tempd2 = tempd2 | 4
*(a2) =ub tempd2

tempd2 =ub *(a2 + [1])
tempd2 = tempd2 | 4
*(a2 + [1]) =ub tempd2

tempd2 =ub *(a2 + [x1])
tempd2 = tempd2 | 4
*(a2 + [x1]) =ub tempd2

tempd2 =ub *(a2 + [x2])

a8 = &*(pba + $pruned_children)
nop
tempd1 =ub *(a8 + x8)
tempd1 = tempd1 + 1
*(a8 + x8) =ub tempd1

br = iprs
tempd2 = tempd2 | 4
*(a2 + [x2]) =ub tempd2

.global $cum_freq
.global $value
.global $low
.global $high
.global $new_decode_symbol
.global big_jump

.align 512

std      .setx2
l        .setd6
save_symbol .seta4

.global $read_more
.global $syms_to_do

big_jump:

*(pba + $syms_to_do) =uh std

a15 = std - 0
br =[eq] done_syms

save_symbol =uw *(pba + $save_symbol)
l = 0

le0 = end_get - 8
lrs0 = std - 1
nop
nop

```

# NAWCWD TP 8442

```

call = $new_decode_symbol
nop
nop

x0 = d5
a0 = &*(xba + $index_to_char)
x1 = 1
tempd1 =ub *(a0 + x0)      ;d1 = index_to_char[sym]

*(save_symbol + x1) =ub tempd1
l = l + 1

call = $new_update_model
nop
d1 = d5

tempd1 = *(pba + $STOP)
tempd1 = tempd1 - 1

lc0 =[eq] 0
nop
nop
nop

end_get:
nop

tempd1 = *(pba + $STOP)
tempd1 = tempd1 - 1

br =[ne] done_syms
nop
nop

;
; syms_to_do = 1 - symbol_index
;
    *(pba + $syms_to_do) =uh 1
    *(pba + $read_more) =ub a15

done_syms:
    a4 = *(sp + 8)
    br = *(sp + 4)
    d6 =sw *(sp + 0)
    d0 = &*(sp += 12)

    .align 512

    .global $fast_cache_syms
    .global $symbol_bit_index
    .global $save_value

```

```

;*****
;* FUNCTION DEF: $fast_cache_syms
;*****

; The assembly language subroutine will realize the following C code

;cache_syms()
;{
;unsigned char i;
;unsigned char sym;
;
;    for (i=0;i<16;i++)
;        save_stop[i] = 1;
;
;    for (i=0;i<16;i++)
;    {
;        symbol_bit_index[i] = bit_index;
;        save_value[i] = value;
;        save_high[i] = high;
;        save_low[i] = low;
;        sym = new_decode_symbol();
;        save_symbol[i] = index_to_char[sym];
;        new_update_model(sym);
;        save_stop[i] = STOP;
;        if (STOP==1)
;            break;
;    }
;
;    STOP = save_stop[0];
;    symbol_bit_index[16] = bit_index;
;    save_value[16] = value;
;    save_high[16] = high;
;    save_low[16] = low;
;    symbol_index = 0;
;
;stophere = 1;
;
;}

$fast_cache_syms:

    d0 = &(sp --= 12)
    *(sp + 8) =w iprs

    *(sp + 4) =w a4
||    *(sp + 0) =w d6

    a0 = &(xba + $save_stop)
    d1 = 0x01010101
    *(a0 + [0]) =uw d1
    *(a0 + [1]) =uw d1
    *(a0 + [2]) =uw d1
    *(a0 + [3]) =uw d1

    le0 = end_for16
    lrs0 = 15
    a4 =uw *(pba + $save_value)
    d6 = 0

    x0 = d6

```

```

a1 = &*(pba + $symbol_bit_index)
d1 =uh *(pba +$bit_index)
*(a1 + [x0]) =uh d1

d1 = *(pba +$value)
*(a4 += 4) = d1

d1 = *(pba +$high)
*(a4 + [16]) = d1

call1 = $new_decode_symbol
d1 = *(pba +$low)
*(a4 + [33]) = d1

x1 = d5
a0 = &*(xba + $index_to_char)
a1 = *(xba + $save_symbol)
x0 = d6
tempd1 =ub *(a0 + x1)      ;d1 = index_to_char[sym]

call1 = $new_update_model
*(a1 + x0) =ub tempd1
d1 = d5

x0 = d6

a1 = &*(pba + $save_stop)
d1 = *(pba +$STOP)
d2 = d1 - 1
lc0 =[eq] 0
*(a1 + [x0]) =ub d1

nop
nop
end_for16:
d6 = d6 + 1

x0 = d6

d1 =ub *(pba + $save_stop)
*(pba + $STOP) = d1

a1 = &*(pba + $symbol_bit_index)
d1 =uh *(pba +$bit_index)
*(a1 + [x0]) =uh d1

d1 = *(pba +$value)
*(a4 += 4) = d1

d1 = *(pba +$high)
*(a4 + [16]) = d1

d1 = *(pba +$low)
*(a4 + [33]) = d1

*(pba + $symbol_index) =ub a15

br = *(sp + 8)

```

```

    d6 = *(sp + 0)
||   a4 = *(sp + 4)
    d0 = &*(sp += 12)

    .global $cum_freq
    .global $value
    .global $low
    .global $high

    .global $new_input_bit
    .global I_DIV_JW2
    .global $new_decode_symbol

    .global $symbols_read

    .align 512

range      .setd0
sym        .setx9
cum        .setd5
tempd1     .setd1
tempd2     .setd2
high       .setd5
low        .setd6
value      .setd7
cum_freq   .seta0
cum_freq_0 .setx2

;*****
;* FUNCTION DEF: $new_decode_symbol *
;*****
$new_decode_symbol:
    a15 = &*(sp --= 12)
    *(sp + 0) =w d7
    *(sp + 4) =w d6

; delete me
    tempd1 =uh *(xba + $symbols_read)
    tempd1 = tempd1 + 1
    *(xba + $symbols_read) =uh tempd1
;

    value =sw *(xba + $value)
    high =sw *(xba + $high)
    low =sw *(xba + $low)

    cum_freq = &*(xba + $cum_freq)

    range = high - low
||   *(sp + 8) =w iprs

    cum = value - low

    range = range + 1

    cum = cum + 1
||   tempd1 =sh *(cum_freq)
    cum = cum * tempd1
||   cum_freq_0 = tempd1

```

# NAWCWD TP 8442

```

call = I_DIV_JW2
tempd1 = cum - 1
tempd2 = range
sym = 0

iloop:
    tempd1 = sh *++cum_freq
    tempd1 = tempd1 - cum
    br =[gt] iloop
    sym = sym + 1
    nop

    high = sh *(cum_freq - [1])
    call = I_DIV_JW2
    tempd1 = high * range
    tempd2 = cum_freq_0

    high = high - 1
    low = high + low      ;save high in low

    tempd1 = sh *(cum_freq)
    call = I_DIV_JW2
    tempd1 = tempd1 * range
    tempd2 = cum_freq_0

    high = low
||   low = high
    tempd1 = sw *(xba + $low)
    low = low + tempd1

for_ever:
    a15 = high - (1\\8)
    br =[lt] done_if
    nop
    nop

    a15 = low - (1\\8)
    br =[lt] else2
    nop
    nop

    value = value - (1\\8)
    br = done_if
    low = low - (1\\8)
    high = high - (1\\8)

else2:
    tempd1 = 0
    a15 = low - (1\\7)
    tempd1 =[lt] 1
    a15 = high - 384
    tempd1 =[ge] 1
    tempd1 = tempd1 - 0
    br =[ne] done_for
    nop
    nop

    value = value - (1\\7)
    low = low - (1\\7)
    high = high - (1\\7)

```

```

done_if:
    low = low << 1
    high = high << 1
    high = high + 1
    call = $new_input_bit
    x1 = d5
    value = value << 1

    br = for_ever
    value = value + d5
    d5 = x1

done_for:
    *(xba + $value) =sw value
    *(xba + $high) =sw high
    *(xba + $low) =sw low

    d5 = x9
    br = *(sp + 8)
    d7 = *(sp + 0)
    || d6 = *(sp + 4)
    d0 = &*(sp += 12)

```

```

.global I_DIV_JW2

```

```

;*****
;* I_DIV.ASM    v1.10    - Integer Divide                                     *
;* Copyright (c) 1993-1995 Texas Instruments Incorporated                     *
;*****

; +-----+
; |      i_div.asm = PP assembly program that is used to return a 32-bit    |
; |      signed integer quotient from 32-bit signed integer                  |
; |      division when called by a C program.                               |
; |                                                                           |
; +-----+

; +-----+
; | 32-bit Signed Integer Word Divide Subroutine :                          |
; |  o Input 32-bit signed integer Operand 1 is in d1 (numerator).           |
; |  o Input 32-bit signed integer Operand 2 is in d2 (divisor).             |
; |  o Output 32-bit signed integer is in d5 (Answer = quotient).            |
; |  o Output 32-bit signed remainder is discarded.                         |
; |  o 0 input divisor produces 0x80000000 output with overflow set.         |
; |  o Quotient = 0x80000000 sets overflow.                                   |
; |  o Number of Stack Words used = 3.                                       |
; |  o MF register is saved.                                                 |
; |                                                                           |
; |  o NOTE: Loop Counter 2 Registers are used but NOT restored !           |
; +-----+

; +-----+
; |      32 bit / 32 bit ==> 32 bit signed quotient                          |
; |      Signed PP Integer Division                                          |
; |      Numerator / Denominator = Quotient + Remainder (discarded)          |
; |      Divide by 0 produces 80000000 and sets sr(V)                       |
; |      Divide Overflow is not possible if Divisor is non-zero,            |
; |      except 80000000/ffffffff = 80000000 will set sr(V).                |
; |      MF register is preserved.                                           |
; +-----+

```



# NAWCWD TP 8442

```

; .ptext      ; PP assembly code

arg1: .setd1  ; input argument 1 = Numerator (32 low bits)
arg2: .setd2  ; input argument 2 = Divisor (32 bits)
ans:.setd5   ; answer = 32 bit signed quotient
Div:.setd3   ; Input Divisor
Num:.setd4   ; Input high Numerator = 0
Tmp:.setd5   ; ALU output for each DIVI

I_DIV_JW2:      ; Signed Word Integer Divide:  Ans = Op1 / Op2

    Div = 0 - | arg2 |      ; negate | divisor |
    || *(sp-=3) = Div      ; || push Div
    br = [z] Div_By_0      ; Divide By 0 ?
    Num = 0                ; high numerator = 0
    || *(sp+[1]) = mf      ; || push mf
    || *(sp+[2]) = Num     ; || push Num
    mf = | arg1 |          ; input lo | numerator |

    lrse2 = 29              ; loop count - 1
    Tmp = divi(Div, Num=Num) ; 1-st divide iterate
    Tmp = divi(Div, Num=Tmp [n] Num) ; 2-nd divide iterate
LoopSW: Tmp = divi(Div, Num=Tmp [n] Num) ; divide iterate 3-32

    ans = mf                ; | ans | = mf
    || Div = *sp++          ; || pop Div
    Num = arg1 ^ arg2       ; quotient sign
    || br = iprs            ; || return
    ans =[n] -ans           ; quotient is negative,
    || mf = *sp++          ; || pop mf
    Num = *sp++             ; pop Num

Div_By_0:      ; Divide By 0 \_____ Optional Error
Div_Ovfl:      ; Divide Overflow /_____ Return Code

    br = iprs          ; return
    || Div = *sp++      ; || pop Div
    mf = *sp++          ; pop mf
    ans = 0 - 1<<31     ; returns 0x80000000, sets sr(V)
    || Num = *sp++      ; || pop Num          ...[END]

tempd4 .setd4
tempd5 .setd5

fred.setd0

.global $new_update_model

;*****
;* FUNCTION DEF: $new_update_model *
;*****
$new_update_model:

    a0 = &*(xba + $cum_freq)
    tempd3 = 0

    tempd2 =sh *a0
    tempd2 = tempd2 - 75
    tempd3 =[ne] 1

```

```

tempd2 =sw *(xba + $No_of_symbols)
a15 = tempd2 - 0
tempd3 =[lt] 1
tempd3 = tempd3 - 0
br =[ne] no_max
a2 = d1
le1 = no_max - 8

tempd5 = tempd2 << 1
|| tempd4 = &*(xba + $freq)
a0 = a0 + tempd5

lrs1 = tempd2
a8 = tempd5 + tempd4
tempd3 = 0

tempd2 =sh *a8
tempd2 = tempd2 + 1
|| *a0-- =h tempd3
tempd2 = tempd2 >>s 1
*a8-- =h tempd2
|| tempd3 = tempd3 + tempd2

no_max:
    d3 = &*(xba + $freq)
    fred = tempd1 << 1
    a0 = fred + d3
    a8 = a0 - 2
    nop

    d4 =sh *a8
    || d3 =sh *a0
    d3 = d3 - d4
    br =[ne] nL11
    le2 = end_while - 8
    nop

freq_eq:
    fred = fred - 2
    || d3 =sh *--a8

    a2 = a2 - 1
    || d4 =sh *--a0
    d3 = d4 - d3
    br =[eq] freq_eq
    d2 =g a2
    a15 = d2 - d1

    br =[ge] nL11
    x0 = &*(xba + $index_to_char)
    x8 = &*(xba + $char_to_index)

    a1 = d1

    a8 =ub *(a2 + x0)
    a9 =ub *(a1 + x0)
    *(a2 + x0) =b a9
    *(a1 + x0) =b a8
    *(a8 + x8) =b a1
    *(a9 + x8) =b a2

```

# NAWCWD TP 8442

```

nL11:
    a15 = a2 - 0
    br = [le] end_while
    d2 = a2 - 1
    d1 = &*(xba + $cum_freq)

    lrs2 = d2
    a1 = fred + d1
    nop

    fred =sh  *--a1
    fred = fred + 1
    *a1 =h  fred

end_while:
    br = iprs
||    d3 =sh  *a0
    d3 = d3 + 1
    *a0 =h  d3

;    branch occurs here

.global $new_input_bit
;*****
;* FUNCTION DEF: $new_input_bit *
;*****
;
; Need to check for STOP in routine that calls this one!!!
;

$new_input_bit:

    d2 =uh  *(xba + $bit_index)
    d4 = d2 >>u 3
||    d3 =uw  *(xba + $byte_stream)

    a0 = d4 + d3
    d0 =uh  *(xba + $T_BYTES)
||    d1 = d2 + 1

    d4 = -(d2&7)
||    d3 =ub  *a0
    d3 = (d3 >>u -d4)

    *(xba + $bit_index) =h  d1
||    d5 = d3 & 1

    d1 = d1 - (d0 << 3)

    br =g iprs
d1 = 1 || d1 = [lt] a15    ;calculate new STOP value
    *(xba + $STOP) =w  d1

.align 512

.global $new_dec_dbits3_ri
.global $save_symbol
.global $save_index

```

```

;*****
;* FUNCTION DEF: $new_dec_dbits3_ri( m) *
;*                               d1 *
;*****

$new_dec_dbits3_ri:

    d0 = &(sp --= 40)
    *(sp + 16) =w iprs

    *(sp + 12) =w a4
||    *(sp + 8) =w d6

    *(sp + 4) =w a12
||    *(sp + 0) =w d7

;fetch symbol from save_symbol[save_index++]

    x0 =uh *(pba + $save_index)
    x1 = x0 + 1
    a0 =uw *(pba + $save_symbol)
    *(pba + $save_index) =uh x1

    symbol =ub *(a0 + x0)

    a1 =uw *(xba + $stats_apx)          ;*(a1) = stats_apx[s*768]
    a0 =uw *(xba + $stats_flag)

    x0 = d1                          ;x0 = m

    ;*(a1 + [x0]) = stats_apx[s*768 + m]
    ;*(a0 + [x0]) = stats_flag[s*768 + m]

    tempd1 =sw *(xba + $BITE)
    tempd1 = 1 << tempd1

    t = symbol - tempd1
    br =[le] sym_gtlad
;    nop
    nop

    tempd1 = 3
    *(a0 + [x0])=ub tempd1

    x8 =uh *(pba + $list_index)
    a15 = x8 - 254
    br =[ge] nosig3d
    tempd1 = x0
    a12 =uw *(pba + $list)
    tempd2 = x8 + 1
    *(a12 + [x8]) =uh tempd1
    *(pba + $list_index) =uh tempd2

nosig3d:

;    stats_apx[m] = ((t*THRESH)>>(BITE-1)) + (THRESH>>BITE);

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

```

# NAWCWD TP 8442

```

tflg = 1 - tempd1

tmask =u tempd2 * t
tempd1 = - tempd1

tempd3 = (tmask >>u -tflg)

br = donead
tempd3 = tempd3 + (tempd2 >>u -tempd1)
*(a1 + [x0]) =h tempd3

sym_gtlad:
t = symbol - 1
br =[le] donead
nop
; nop

tempd1 = 1
*(a0 + [x0])=ub tempd1

x8 =uh *(pba + $list_index)
a15 = x8 - 254
br =[ge] nosig4d
tempd1 = x0
a12 =uw *(pba + $list)
tempd2 = x8 + 1
*(a12 + [x8]) =uh tempd1
*(pba + $list_index) =uh tempd2

nosig4d:

; stats_apx[m] = - ((t*THRESH)>>(BITE-1)) - (THRESH>>BITE);

tempd1 =sw *(xba + $BITE)
tempd2 =uh *(xba + $THRESH)

tflg = 1 - tempd1

tmask =u tempd2 * t
tempd1 = - tempd1

tempd3 = -(tmask >>u -tflg)
tempd3 = tempd3 - (tempd2 >>u -tempd1)
*(a1 + [x0]) =h tempd3

donead:
a12 =w *(sp + 4)
a4 =w *(sp + 12)

br = *(sp + 16)

d6 =sw *(sp + 8)
|| d7 =sw *(sp + 0)
d0 = &*(sp += 40)

; branch occurs here

.global $quick_clear

```

# NAWCWD TP 8442

\$quick\_clear:

```
a0 =uw *(xba + $stats_apx)
lrse0 = 767
a8 =uw *(xba + $stats_flag)
d1 = 0
```

```
*(a0 ++=[1]) = d1
|| *(a8 ++=[1]) =uh d1
```

```
br = iprs
nop
nop
```

# NAWCWD TP 8442

```

/*****
**
** faster_ar.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains fast_arith_decode2 and the do_subdec subroutine.
**
*****/

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

#define CACHE
#define ASSEM

extern unsigned char emubrk;

extern unsigned char *stats_flag;
extern int STOP;
extern unsigned short syms_to_do;
extern unsigned short *save_array;
extern unsigned char read_more;
extern unsigned short left_off;
extern int FIRST_DEC;
extern unsigned char RB_DEC;
extern unsigned char PASS_DEC;
extern unsigned short THRESH;
extern unsigned short *list;
extern unsigned short list_index;
extern unsigned char index_to_char[Max_No_of_symbols+1];
extern short *stats_apx;
extern unsigned short symbols_read;
extern int BITE;

extern unsigned char pruned_children[8];
extern unsigned char total_links;
extern unsigned char link_list[6];

extern unsigned short bit_index;

extern int value;          /* Currently-seen code value */
extern int low, high;      /* Ends of current code region */

extern int *save_value;
extern int *save_high;
extern int *save_low;

extern unsigned short symbol_bit_index[17]; /* bit_index value after each
symbol has been read into the cache */
extern unsigned char save_stop[16];        /* STOP value after each symbol has
been read into the cache */
extern unsigned char symbol_index;         /* index to next symbol in the cache
*/

void do_subdec();

```

```

/*****
/*
/* fast_arith_decode2 will be compiled with the -o2 (optimization full) */
/* command from the compiler to optimize this code for fast operation. */
/*
/* The routine scans thru the zero-tree and turns symbols into */
/* coefficients where needed (pruning trees as we go...) */
/*
/*****/

void fast_arith_decode2()
{
    int i,m,c,s;

#ifdef CACHE
    fast_cache_syms();
#endif

    /* DECODE SIGNIFICANCE MAP */

    for (s=0;s<6;s++) pruned_children[s] = 0;

    for (s=0;s<6;s++)
        dec_dbits_fix(s);

    m = 0;
    for (s=0;s<6;s++)
    {
        if (pruned_children[s] < 1)
        {
            link_list[m++] = s;
            pruned_children[s] = 0;
        }
        else
        {
            pruned_children[s] = 255;
            for (i=(s*256)+1;i<((s*256)+4);i++)
            {
                stats_flag[i] = stats_flag[i] & 251;
            }
        }
    }

    total_links = m;

    for(m=1;m<4;m++)
    {
        c=4*m;

        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
#ifdef ASSEM
            if ((stats_flag[m + s*256]&4)==4)
                new_dec_dbits_nri(m,c,2,s);
            else if ((stats_flag[m + s*256]&5)==0)
                new_dec_dbits_ri(m,c,2,s);

#else
            dec_dbits(m,c,c+1,c+2,c+3,s);
#endif
        }
    }
}

```



```

        if (STOP==1)
            break;
    }
    if (STOP==1)
        break;
    }

m = 0;
for (s=0;s<6;s++)
{
    if (pruned_children[s] < 3 )
    {
        link_list[m++] = s;
        pruned_children[s] = 0;
    }
    else
    {
        if (pruned_children[s] != 255)
        {
            for (i=(s*256+4);i<(s*256+16);i++)
            {
                stats_flag[i] = stats_flag[i] & 251;
            }
        }
        pruned_children[s] = 255;
    }
}

total_links = m;

for(m=4;m<16;m++)
{
    c = 8*(m/2) + 2*(m%2);
    for (i=0;i<total_links;i++)
    {
        s = link_list[i];
#ifdef ASSEM
        if ((stats_flag[m + s*256]&4)==4)
            new_dec_dbits_nri(m,c,4,s);
        else if ((stats_flag[m + s*256]&5)==0)
            new_dec_dbits_ri(m,c,4,s);
#else
        dec_dbits(m,c,c+1,c+4,c+5,s);
#endif

        if (STOP==1)
            break;
    }

    if (STOP==1)
        break;
}

m = 0;
for (s=0;s<6;s++)
{
    if (pruned_children[s] < 12 )
    {
        link_list[m++] = s;
    }
}

```

```

    pruned_children[s] = 0;
  }
  else
  {
    if (pruned_children[s] != 255)
    {
      for (i=(s*256+16);i<(s*256+64);i++)
      {
        stats_flag[i] = stats_flag[i] & 251;
      }
    }
    pruned_children[s] = 255;
  }
}

total_links = m;

for(m=16;m<64;m++)
{
  c = 16*(m/4) + 2*(m%4);
  for (i=0;i<total_links;i++)
  {
    s = link_list[i];
#ifdef ASSEM
    if ((stats_flag[m + s*256]&4)==4)
      new_dec_dbits_nri(m,c,8,s);
    else if ((stats_flag[m + s*256]&5)==0)
      new_dec_dbits_ri(m,c,8,s);

#else
    dec_dbits(m,c,c+1,c+8,c+9,s);
#endif

    if (STOP==1)
      break;
  }

  if (STOP==1)
    break;
}

/* Eliminate if NSCALES = 3 */

m = 0;
for (s=0;s<6;s++)
{
  if (pruned_children[s] < 48 )
  {
    link_list[m++] = s;
    pruned_children[s] = 0;
  }
  else
  {
    if (pruned_children[s] != 255)
    {
      for (i=(s*256+64);i<(s*256+256);i++)
      {
        stats_flag[i] = stats_flag[i] & 251;
      }
    }
    pruned_children[s] = 255;
  }
}

```

```

    }
}

total_links = m;

emubrk = 2;

if ((STOP==0) && (total_links!=0))
{

read_more = 0;
left_off = 0;

pre_dbits2();

while (syms_to_do>0)
{

for (i=0;i<syms_to_do;i++)
{
m = save_array[i];
new_dec_dbits3_ri(m);
}

syms_to_do = 0;

if (read_more == 1)
{
pre_dbits2();
}
}

else
{
if (total_links == 0)
{
bit_index = symbol_bit_index[symbol_index];
value = save_value[symbol_index];
high = save_high[symbol_index];
low = save_low[symbol_index];
}
}

emubrk = 3;

do_subdec();

}

void do_subdec()
{
int t,tt,sym,symbol,i,m,j;

/* restore pointer to start getting new symbols for a different symbol size */

start_model(2);

if (STOP != 1)
PASS_DEC+=BITE;

```

```

t = THRESH>>BITE;
THRESH = THRESH>>BITE;

if (PASS_DEC==BITE)
    BITE = RB_DEC;
else
    BITE = 1;

#ifdef NO_MULTI_BITPLANE
BITE = 1;
#endif

/* DECODE RESOLUTION INCREASE */

if (STOP==0)
{
    for(i=0;i<list_index;i++)
    {
        m = list[i];
        tt = t;
        for(j=0;j<BITE;j++)
        {
            sym = new_decode_symbol();
            symbol = index_to_char[sym];
            new_update_model(sym);

            stats_apx[m] += (((stats_flag[m]&2)>0) ? 1:-1)*(((1&symbol)>0) ? 1:-
1)*((tt+1)/2);
            tt = tt/2;
            if (STOP==1)
                break;
        }
        if (STOP==1)
            break;
    }
}

start_model(1<<(BITE+1));
}

```

# NAWCWD TP 8442

```

/*****
*
*   Function : form_img
*   Args      : s - subblock number (0 - 5)
*               p - pair number (0,1)
*
*   Description:
*       form_img will be called to copy the coefficients from one
*       section of memory in zero-tree format to another place
*       in memory in in-place format
*
*   Return Values:
*       None
*
*****/

#define NSCALES 5

extern short *coeff_block;
extern short *stats_apx;

/*****
*                               FORM_IMG
*                               *****/

void form_img2(s,p)
    int s,p;
    {
        int k,j,l,i,ty,tx;

        i = s * 256;
        for(k=NSCALES-2;k>0;k--)
        {
            if (k==(NSCALES-2))
                for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
                    for(l=0;l<(1<<NSCALES);l+=(2<<k))
                        coeff_block[p*512+j*32+1] = stats_apx[i++];

            for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
                    coeff_block[p*512+j*32+1] = stats_apx[i++];

            for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=0;l<(1<<NSCALES);l+=(2<<k))
                    coeff_block[p*512+j*32+1] = stats_apx[i++];

            for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
                for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
                    coeff_block[p*512+j*32+1] = stats_apx[i++];

        }

        for(j=0;j<(1<<(NSCALES-1));j++)
            for(l=1;l<(1<<NSCALES);l+=2)
                coeff_block[p*512+j*32+1] = 0;

    }

```

# NAWCWD TP 8442

```

/*****
**
** hvdec.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $subdec_vert
** $subdec_horiz
**
*****/

.global $subsyn_vert
.global $subsyn_horiz

.align 2048

/*****
*
* Function : $subsyn_horiz
* Args : outerloop,innerloop
* Passed in : (d1 ,d2)
*
* Description:
* outerloop - the width of the coefficient patch
*
* innerloop - the height of the coefficient patch
*
* $subsyn_horiz will be called to perform the wavelet
* composition in the horizontal direction.
*
* Return Values:
* None
*
*****/

$subsyn_horiz:

    lctl = 0x0 ;reset looping capability

;set loop reload, counter
    lr0 = d1 - 1
    lr1 = d2 - 2

    a4 = d2 ; innerloop

    d7 = 0 ; k = 0

; Set up loop to iterate ((128 >> scale) - 2) times
    le1 = InnerLoopEnd2 ;
    ls1 = InnerLoop2 ;

    le0 = OuterLoopEnd2 ;
    ls0 = OuterLoop2 ;

    nop

    lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

    nop
    nop

```

# NAWCWD TP 8442

```

OuterLoop2:
    d3 = a4
    d3 = d3 * d7
    d3 = d3 << 1
    x0 = d3
    nop
    a0 =h &*(dba + [x0])
    nop

; >>>> new          img[k][0] = (img[k][0]-img[k][index])>>1;
;                   [a0]          [a0]          [a0+1]
    d2 =sh *(a0)
    d4 =sh *(a0 + [1])
    d3 = d2 - d4
    d3 = (d3 >> 1)
    *(a0) =h d3

    nop

InnerLoop2:
; save first stored variable (d3?)
; img[k][2*1+index] -= ((img[k][2*1]+img[k][2*1+2*index])>>1);
; img[k][2*1] = (img[k][2*1]>>1) - ((img[k][2*1+index] + img[k][2*1-index])>>2);

    d3 =sh *(a0 + [1])
    d4 =sh *(a0 + [3])

    d2 =sh *(a0 + [2])
    || d5 = d3 + d4

    d2 = d2 >> 1
    d2 = d2 - (d5 >>s 2)

    *(a0 + [2]) =h d2

; img[k][2*1] = (img[k][2*1]<<1) + ((img[k][2*1+index]+img[k][2*1-index])>>1);
; img[k][2*1-index] += ((img[k][2*1] + img[k][2*1-2*index])>>1);

    d4 =sh *(a0++=[1])
    d5 = d2 + d4
    d3 = d3 + (d5 >>s 1)

    *(a0++=[1]) =h d3

InnerLoopEnd2:
    nop

; >>>>          img[k][XSIZE-index] -= img[k][XSIZE-2*index]; old
; >>>>          img[k][XSIZE-index] += img[k][XSIZE-2*index]; new

    d2 =sh *(a0 )
    d3 =sh *(a0 + [1])
    d2 = d2 + d3
    *(a0 + [1]) =h d2

OuterLoopEnd2:
    d7 = d7 + 1

    br = iprs

    nop
    nop

```

```

/*****
*
*   Function   : $subsyn_vert
*   Args      : outerloop, innerloop
*   Passed in : (d1      , d2)
*
*   Description:
*       outerloop - the width of the coefficient patch
*
*       innerloop - the height of the coefficient patch
*
*       $subsyn_vert will be called to perform the wavelet
*       composition in the vertical direction.
*
*   Return Values:
*       None
*
*****/

$subsyn_vert:
    lctl = 0x0 ;reset looping capability

    lr0 = d1 - 1      ;outerloop
    lr1 = d2 - 2      ;innerloop
    a4 = d1

    d7 = 0            ; k = 0

;   Set up zero-overhead loops
    le1 = InnerLoopEnd      ;
    ls1 = InnerLoop         ;

    le0 = OuterLoopEnd
    ls0 = OuterLoop
    nop
    lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

    d3 = a4
    d4 = d3 + d3
    x1 = d4
    d4 = d4 + d3
    x2 = d4

OuterLoop:
;>>>>          img[index][k]      =      (img[index][k]>>1)      -
((img[0][k]+img[2*index][k])>>2);

    nop
    x0 = d7
    nop
    a0 =h &*(dba + [x0])
    nop
    x0 = a4

;>>>>          img[0][k] += img[index][k];
;>>>>          img[0][k] -= img[index][k];

    d3 =h *(a0)
    d4 =h *(a0 + [x0])
    d3 = d3 - d4
    *(a0) =h d3

```



# NAWCWD TP 8442

nop

InnerLoop:

```
;img[2*1+index][k] = (img[2*1+index][k]>>1) -
((img[2*1][k]+img[2*1+2*index][k])>>2);
;img[2*1][k] -= ((img[2*1+index][k] + img[2*1-index][k])>>1);

d3 =sh *(a0 + [x0])
d4 =sh *(a0 + [x2])

d2 =sh *(a0 + [x1])
|| d5 = d3 + d4

d2 = d2 - (d5 >> 1)

*(a0 + [x1]) =h d2

;img[2*1][k] += ((img[2*1+index][k]+img[2*1-index][k]+2)>>1);
;img[2*1-index][k] = (img[2*1-index][k]<<1) + ((img[2*1][k] + img[2*1-
2*index][k])>>1);

d4 =sh *(a0++=[x0])
d5 = d2 + d4
d3 = d3 << 1
d3 = d3 + (d5 >> 1)

*(a0++=[x0]) =h d3
```

InnerLoopEnd:

nop

```
;>>>> img[YSIZE-index][k] = (img[YSIZE-index][k]-img[YSIZE-
2*index][k])>>1;
;>>>> img[YSIZE-index][k] = (img[YSIZE-index][k]<<1) + img[YSIZE-
2*index][k];

d2 =sh *(a0)
d3 =sh *(a0 + [x0])
d3 = d3 << 1
d2 = d2 + d3
*(a0 + [x0]) =h d2
```

OuterLoopEnd:

d7 = d7 + 1

br = iprs

nop

nop

```

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

/* #define NO_MULTI_BITPLANE */

#define CACHE

#define ASSEM

#define XSIZE 512
#define YSIZE 256
#define NSCALES 5
#define AX 16 /* = XSIZE/BS */
#define AY 8 /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

/* REAL-TIME: 2 Zerotrees/partition, last subband truncated */

int stophere;

extern shared int global_maxval;

extern short *coeff_block;
extern short *stats_apx;
extern unsigned char *stats_flag;
extern unsigned char *byte_stream;

extern char *tbuf;

extern unsigned short *list;
unsigned short list_index;

short BYTE_TOTAL;

extern int No_of_chars;

extern int EOF_symbol; /* Index of EOF symbol */

extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short int cum_freq[Max_No_of_symbols+1];

extern int buffer_index; /* JCW */

extern unsigned short T_BYTES;

extern int STOP;
extern int BYTE_CNT;
extern unsigned short THRESH;
extern int BITE;
extern int SIG_COEF;
/* extern int SIG[15]; */

extern short maxval;

extern unsigned short bit_index;

```

# NAWCWD TP 8442

```

extern int FIRST_DEC;

unsigned char RB_DEC;

unsigned char PASS_DEC;

extern unsigned short *ALLOC;

void start_decoding();

void new_dec_dbits_ri (int m, int c1, int c3, int s);
void new_dec_dbits_nri(int m, int c1, int c3, int s);

void form_img2(int s, int p);

/* The bit buffer */

unsigned char buffer;      /* Bits buffered for output */
unsigned char bits_to_go; /* # bits free in buffer */

/* Current state of the decoding */

int value;      /* Currently-seen code value */
int low, high;  /* Ends of current code region */

unsigned char emubrk;

unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

unsigned char *save_symbol;      /* cache for symbols */
unsigned short save_index;
unsigned short syms_to_do;

unsigned short *save_array;
unsigned char read_more;
unsigned short left_off;

int *save_value;
int *save_high;
int *save_low;

unsigned short symbols_read;

unsigned char pass_num;

/* needed for cacheing of symbols */

unsigned short symbol_bit_index[17]; /* bit_index value after each symbol has
been read into the cache */
unsigned char save_stop[16];        /* STOP value after each symbol has been read
into the cache */
unsigned char symbol_index;          /* index to next symbol in the cache */

unsigned char pruned_children[8];
unsigned char total_links;
unsigned char link_list[6];

```

```

/* *****
                                DEC_DBITS
***** */
void dec_dbits_fix(s)
    int s;

{
    int sym, symbol, t;
    int index;

    index = (s*256);

    if (((stats_flag[index]&1)==0)&&((stats_flag[index]&4)==0))
    {

/* delete me */
symbols_read += 1;

#ifdef CACHE
        STOP = save_stop[symbol_index];
        symbol = save_symbol[symbol_index++];
#else
        sym = new_decode_symbol();
        symbol = index_to_char[sym];
        new_update_model(sym);
#endif

        if (symbol > (1<<BITE))
        {
            stats_flag[index] = 3;
            if (list_index < 254)
                list[list_index++] = index;
            t = symbol-(1<<BITE);
            stats_apx[index] = ((t*THRESH)>>(BITE-1))+(THRESH>>BITE);
        }
        else if (symbol > 1)
        {
            stats_flag[index] = 1;
            if (list_index < 254)
                list[list_index++] = index;
            t = symbol-1;
            stats_apx[index] = -((t*THRESH)>>(BITE-1))-(THRESH>>BITE);
        }
        else if (symbol==0)
        {
            pruned_children[s] = 1;
            stats_flag[index+1] = stats_flag[index+1] | 4;
            stats_flag[index+2] = stats_flag[index+2] | 4;
            stats_flag[index+3] = stats_flag[index+3] | 4;
        }
    }
    else if ((stats_flag[index]&4)!=0)
    {
        pruned_children[s] = 1;
        stats_flag[index] = stats_flag[index] & 251;
        stats_flag[index+1] = stats_flag[index+1] | 4;
        stats_flag[index+2] = stats_flag[index+2] | 4;
        stats_flag[index+3] = stats_flag[index+3] | 4;
    }
    else
        stats_flag[index] = stats_flag[index] & 251;

```

# NAWCWD TP 8442

```

    }

/* *****
                                     P_DEC
***** */

void p_dec()
{
    int k, kk, l, i, j, X, Y, sum, bite;

/* Main Loop: Continue until stop condition is reached */

    FIRST_DEC = tbuf[0];

    if (FIRST_DEC == 6)
    {
        bite = 3;
        RB_DEC = 3;
    }
    else if (FIRST_DEC == 5)
    {
        bite = 3;
        RB_DEC = 2;
    }
    else if (FIRST_DEC == 4)
    {
        bite = 2;
        RB_DEC = 2;
    }
    else
    {
        bite = FIRST_DEC;
        RB_DEC = 1;
    }

/* tell MP this PP is ready to decode the bit stream */

    asm("    x2 = 0x00002100");
    asm("    cmnd = x2");

/* wait for all PPs to catch up (signed by int from MP) */

while((INTFLG & (1 << 20)) == 0);
    INTFLG = 1 << 20;

/* read in global_maxval for this image (from the MP) */

    maxval = 1 << global_maxval;

    while (*pp_stop_encode == 0)
    {

        BYTE_CNT = 0;
        buffer_index = 0;
        STOP = 0;
        list_index = 0;
        PASS_DEC = 0;
        THRESH = maxval;
        BITE = bite;
        bit_index = 0;
    }

```

```

#ifdef NO_MULTI_BITPLANE
BITE = 1;
#endif

    start_model(1<<(BITE+1));

    start_inputting_bits();

/* wait for new bit stream */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

if (*pp_stop_encode == 0)
{
    T_BYTES = ALLOC[0];

    quick_clear();

    stats_flag[1536] = 0;

    start_decoding();

    pass_num = 0; /* delete me */

    while (STOP == 0)
    {
        symbols_read = 0;
        fast_arith_decode2();
    }

    form_img2(0,0);
    form_img2(1,1);

    asm("    x2 = 0x00002100"); /* tell MP done with this block of
coefficients */
    asm("    cmnd = x2");

    while((INTFLG&(1<<20))==0); /* wait for MP to read coefficients */
    INTFLG = 1<<20;

    form_img2(2,0);
    form_img2(3,1);

    asm("    x2 = 0x00002100"); /* tell MP done with this block of
coefficients */
    asm("    cmnd = x2");

    while((INTFLG&(1<<20))==0); /* wait for MP to read coefficients */
    INTFLG = 1<<20;

    form_img2(4,0);
    form_img2(5,1);

    asm("    x2 = 0x00002100"); /* tell MP done with this block of
coefficients */
    asm("    cmnd = x2");

    }
}
}

```

# NAWCWD TP 8442

```

/* BIT INPUT ROUTINES */
/* INITIALIZE BIT INPUT */

start_inputting_bits()
{
    bits_to_go = 0;
}

/* START DECODING A STREAM OF SYMBOLS */

void start_decoding()
{
    int i,t;

    value = 0;          /* Fill code value */
    for(i=1;i<=Code_value_bits;i++)
    {
        t = new_input_bit();
        value = 2*value+t;
    }
    low = 0;
    high = Top_value;
}

/* DECODE THE NEXT SYMBOL */

int decode_symbol()
{
    long range;
    int cum,t; /* Cumulative freq calculated */
    int symbol; /* Decoded symbol */

    range = (long) (high-low)+1;
    cum = ((long) (value-low)+1)*cum_freq[0]-1)/range;
    for(symbol=1;cum_freq[symbol]>cum;symbol++); /* Find symbol */

    /* Narrow ranges */
    high = low + (range*cum_freq[symbol-1])/cum_freq[0]-1;
    low = low + (range*cum_freq[symbol])/cum_freq[0];

    for(;;)
    {
        if (high<Half)
        {
            /* Do nothing */
        }
        else if (low>=Half)
        {
            value -= Half;
            low -= Half;
            high -= Half;
        }
        else if (low>=First_qtr && high<Third_qtr)
        {
            value -= First_qtr;
            low -= First_qtr;
            high -= First_qtr;
        }
        else break;
    }
}

```

```

        low = 2*low;
        high = 2*high+1;
        t = new_input_bit();
        value = 2*value+t;
    }
    return symbol;
}

cache_syms()
{
    unsigned char i;
    unsigned char sym;

    for (i=0;i<16;i++)
        save_stop[i] = 1;

    for (i=0;i<16;i++)
    {
        symbol_bit_index[i] = bit_index;
        save_value[i] = value;
        save_high[i] = high;
        save_low[i] = low;
        sym = new_decode_symbol();
        save_symbol[i] = index_to_char[sym];
        new_update_model(sym);
        save_stop[i] = STOP;
        if (STOP==1)
            break;
    }

    STOP = save_stop[0];
    symbol_bit_index[16] = bit_index;
    save_value[16] = value;
    save_high[16] = high;
    save_low[16] = low;
    symbol_index = 0;

    stophere = 1;
}

pre_dbits2()
{
    int sym;
    int not_done;
    int i,j,l;
    int loop_counter;
    unsigned char num_in_cache;
    unsigned char tstop;
    unsigned char more_to_do;
    unsigned char temp_index;
    unsigned char cache_previously_exhausted;

    cache_previously_exhausted = read_more;

    quick_syms();

    if ((syms_to_do == 0)&&(cache_previously_exhausted == 0))
    {
        bit_index = symbol_bit_index[symbol_index];

```



```

value = save_value[symbol_index];
high = save_high[symbol_index];
low = save_low[symbol_index];
}

if (syms_to_do>0)
{
stophere = 2;

if (cache_previously_exhausted == 0)
{

;
; /* std always > 0, if we have gotten here*/

num_in_cache = 16 - symbol_index;

save_index = symbol_index;

if ( syms_to_do > num_in_cache)
    tstop = save_stop[15];
else
    tstop = save_stop[symbol_index + syms_to_do - 1];

if (tstop == 1)
{
    for (i=0;i<num_in_cache;i++)
        if (save_stop[symbol_index + i] == 1)
            break;

    syms_to_do = i + 1;
    read_more = 0;
    STOP = 1;
    return(0);
}

/* do we already have all the symbols we need in the cache */
if (syms_to_do<=num_in_cache)
{
    bit_index = symbol_bit_index[symbol_index + syms_to_do];
    value = save_value[symbol_index + syms_to_do];
    high = save_high[symbol_index + syms_to_do];
    low = save_low[symbol_index + syms_to_do];
    read_more = 0;
    return(0);
}

/* if we got here we need to read more symbols */

more_to_do = syms_to_do - num_in_cache;

i = 0;
while ((STOP==0) && (i<more_to_do))
{
    sym = new_decode_symbol();
    save_symbol[16 + i] = index_to_char[sym];
    new_update_model(sym);
    i = i + 1;
}

```

```
if (STOP==1)
{
    syms_to_do = i + num_in_cache;
    read_more = 0;
}
}
else
{
    save_index = 0;
    i = 0;
    while ((STOP==0) && (i<syms_to_do))
    {
        sym = new_decode_symbol();
        save_symbol[i] = index_to_char[sym];
        new_update_model(sym);
        i = i + 1;
    }

    if (STOP==1)
    {
        syms_to_do = i;
        read_more = 0;
    }
}
}
```

# NAWCWD TP 8442

```

/*****
**
** main.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Main PP Program that calls all the other routines to perform the wavelet
** bitstream decoding and wavelet composition.
**
*****/

#include "modlp.h"

#define XCOLS 8      /* Max Columns that can be held in internal memory */
#define YROWS 4      /* Max Columns that can be held in internal memory */

#define XSIZE 512    /* Max Image Size */
#define YSIZE 240     /* Max Image Height */
#define ML 50        /* Max order of filter allowed */
#define NSCALES 4
#define AX 16 /* = XSIZE/BS */
#define AY 8  /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

#define DECODE_STREAM
#define DECODE

#define DISPLAY

    unsigned short T_BYTES;
    unsigned char *pic;
    short *img;

    short *stats_val;
    short *stats_apx;
    unsigned char *stats_flag;
    unsigned int *stomp_flag;
    unsigned int *stomp_ztl;
    unsigned char *byte_stream;
    short *coeff_block;
    unsigned short *ztl;
    unsigned char *tbuf;
    unsigned short *ALLOC;

    unsigned short *list;

extern unsigned short *save_array;
extern unsigned short *save_symbol;
extern int *save_value;
extern int *save_high;
extern int *save_low;

int No_of_chars;

int EOF_symbol; /* Index of EOF symbol */

int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

unsigned char char_to_index[Max_No_of_chars]; /* JCW old version was int */
unsigned char index_to_char[Max_No_of_symbols+1]; /* JCW old version was int */

```

# NAWCWD TP 8442

```

short int cum_freq[Max_No_of_symbols+1]; /* JCW old version was int */

int buffer_index; /* JCW */

extern void subdec_vert(int outerloop,int innerloop);
extern void subdec_horiz(int outerloop,int innerloop);

extern int calc_n_sub_mean(int oldmean);
extern void add_n_conv8(int oldmean);

extern void subsyn_vert(int outerloop,int innerloop);
extern void subsyn_horiz(int outerloop,int innerloop);

extern int whoami();

/* ***** MAIN PROGRAM ***** */

cregister extern volatile unsigned int INTFLG;

short column_outerloop[5] = { 8, 16, 32, 16, 8 };
short row_outerloop[5]    = { 4, 8, 16, 8, 4 };

short column_innerloop[5] = { 120, 60, 30, 15, 8 };
short row_innerloop[5]    = { 256, 128, 64, 32, 16 };

unsigned char vert_loop_index[6] = { 16, 4, 1, 1, 1 };
unsigned char horiz_loop_index[6] = { 15, 5, 1, 1, 1 };

short maxval;

extern shared int mean_pp[4];
extern shared int global_mean;

unsigned char *passes_d;

extern unsigned char PASS_ENC,PASS_DEC;

int STOP;
int BYTE_CNT;
unsigned short THRESH;
int BITE;
int SIG_COEF;

int FIRST_ENC,FIRST_DEC;
int FCNT_ENC,FCNT_DEC;

/* delete me */
/*extern unsigned char symbols_in_pass[];
extern unsigned char min_in_pass[];
extern unsigned char max_in_pass[]; */

```

```

main()
{

    int k,l;
    int mean;

    /* initialize pic[], img[], stats_val[], stats_flag[], byte_stream[], and
    coeff_block[] pointers */

    asm("    d1 = &(dba)");
    asm("    *(xba+$pic) = d1");
    asm("    *(xba+$img) = d1");
    asm("    *(xba+$stats_val) = d1");
    asm("    *(xba+$stats_apx) = d1");
    asm("    d1 = &(dba + 0x8000)");
    asm("    *(xba+$stats_flag) = d1");
    asm("    *(xba+$stomp_flag) = d1");
    asm("    *(xba+$coeff_block) = d1");
    asm("    x0 = 0x8602");
    asm("    nop");
    asm("    d1 = &(dba + x0)");
    asm("    *(xba+$list) = d1");
    asm("    d1 = &(dba + 0xc00)");
    asm("    *(xba+$ztl) = d1");
    asm("    *(xba+$stomp_ztl) = d1");
    asm("    *(xba+$save_array) = d1");
    asm("    d1 = &(dba + 0xe00)");
    asm("    *(xba+$save_symbol) = d1");
    asm("    d1 = &(dba + 0xf00)");
    asm("    *(xba+$save_value) = d1");
    asm("    d1 = &(dba + 0xf44)");
    asm("    *(xba+$save_high) = d1");
    asm("    d1 = &(dba + 0xf88)");
    asm("    *(xba+$save_low) = d1");
    asm("    d1 = &(pba + 0x630)");
    asm("    *(xba+$byte_stream) = d1");
    asm("    d1 = &(pba + 0x620)");
    asm("    *(xba+$passes_d) = d1");
    asm("    d1 = &(pba + 0x5fc)");
    asm("    *(xba+$tbuf) = d1");
    asm("    d1 = &(pba + 0x600)");
    asm("    *(xba+$ALLOC) = d1");

    FIRST_ENC = 4;
    FIRST_DEC = 4;

    FCNT_ENC = 0;
    FCNT_DEC = 0;

    /* Clear the message interrupt flag that comes from the MP, just in case */

    INTFLG = 1<<20;

    while (1)
    {

#ifdef DECODE_STREAM
        p_dec();
#endif

#ifdef DECODE

```

```

/* Reconstruct image */

for(k=NSCALES;k>=0;k--)
{
  for(l=0;l<horiz_loop_index[k];l++)
  {
    /* wait for new pixels */

    while((INTFLG&(1<<20))==0);

    INTFLG = 1<<20;

    subsyn_horiz(row_outerloop[k],row_innerloop[k]);

    if (k==0)
      add_n_conv8(global_mean);

    /* tell MP done with this block of pixels */

    asm("    d7 = 0x00002100");
    asm("    cmnd = d7");

  }

  if (k!=0)
    for(l=0;l<vert_loop_index[k-1];l++)
    {
      /* wait for new pixels */

      while((INTFLG&(1<<20))==0);

      INTFLG = 1<<20;

      subsyn_vert(column_outerloop[k-1],column_innerloop[k-1]);

      /* tell MP done with this block of pixels */

      asm("    d7 = 0x00002100");
      asm("    cmnd = d7");

    }
}

/* extra synchronizing wait for interrupt */

while((INTFLG&(1<<20))==0);
INTFLG = 1<<20;

#endif

}

}

```

# NAWCWD TP 8442

```

/*****
**
** modlp.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains start_model subroutine. This subroutine initializes
** the translation tables and the frequency counters for the
** arithmetic coder.
**
*****/

/* ADAPTIVE SOURCE MODEL */

#include "modlp.h"
#include "arith.h"

short int freq[Max_No_of_symbols+1]; /* Symbol frequencies */

extern int No_of_chars;
extern int EOF_symbol; /* Index of EOF symbol */
extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short int cum_freq[Max_No_of_symbols+1];

extern int low,high;
extern short bits_to_follow;
extern unsigned char bits_to_go;
extern unsigned char buffer;
extern int BYTE_CNT;
extern unsigned short T_BYTES;
extern unsigned char *byte_stream;
extern int buffer_index;
extern short BYTE_TOTAL;
extern int STOP;

/* INITIALIZE THE MODEL */

void start_model(nchars)
    int nchars;
{
    int i;

/* Initialize number of chars */

    No_of_chars = nchars;
    No_of_symbols = nchars;

/* Setup translation tables */

    for(i=0;i<No_of_chars;i++)
    {
        char_to_index[i] = i+1;
        index_to_char[i+1] = i;
    }

```

```
/* Initialize frequency counts */  
for(i=0;i<=No_of_symbols;i++)  
{  
    freq[i] = 1;  
    cum_freq[i] = No_of_symbols-i;  
}  
freq[0] = 0;  
}
```



## NAWCWD TP 8442

```
/* INTERFACE TO THE MODEL */  
  
/* Set of symbols that may be encoded */  
  
#define Max_No_of_chars 16  
#define Max_No_of_symbols 17  
  
/* Cumulative Frequency Table */  
  
#define Max_frequency 75 /* 16383 Maximum allowed frequency cnt  $2^{14}-1$  */
```

# NAWCWD TP 8442

```

/*****
**
** mp.c (MP Program)
**
** Written by Jim Witham, Code 472330D, 927-1440
**
** MP Program that orchestrates data movement and kicks off PP's to implement
** the balanced wavelet algorithm written by Chuck Creusere.
**
*****/

#include <stdlib.h>
#include <stdio.h>
#include <mvp.h>

#include <mvp_hw.h>
#include <mp_ptreq.h>

#include "icl.h"
#include "vil24.h"
#include "vol.h"
#include "bgl.h"
#include "pcic80.h"
#include "cil.h"

/* define compression ratio. Ratio is actually #:1 compression */

#define HOST
/* #define variable_compression */
#define compression 40
#define headersize 0

#define DECODE_STREAM
#define DECODE

#define DISPLAY

/* #define SHOW_WAITFORHOST */
/* #define SHOW_DECODETIME */
/* #define SHOW_FRAME TIME */
#define SHOW_DSPUTIL
#define SHOW_FRAMERATE

/* #define INTER_DISPLAY */

/* #define OLD_DISPLAY */

/* #define DEBUG */

#define XSIZE 512
#define YSIZE 240

#define AX 16
#define AY 15

```

```

/*
** Define a bit mask for host request bit
*/
#define SigBit(x)      (((UINT32)1)<<(x))
#define HostRequestBitMask SigBit(8)
#define C80ReadyBit 8
#define C80DoneBit 9

#define d_buffers 6

/*****
extern int *number_pixels;
extern int *logo;
extern int *frame_rate;
extern int *dsp_util;

int encode_time,decode_time;
unsigned int time_in_wait;

unsigned int start_time;
/* int subblock_time[16]; */

/* int e[6]; */
int time_colin,time_colout;
int last_etime,last_dtime;

#pragma DATA_SECTION(mean_pp,"sh_vars")
#pragma DATA_SECTION(global_mean,"sh_vars")

    shared int mean_pp[4];
    shared int global_mean;

#pragma DATA_SECTION(local_maxval,"sh_vars")
#pragma DATA_SECTION(global_maxval,"sh_vars")

    shared int local_maxval[4];
    shared int global_maxval;

/* UINT32 buf_addr[d_buffers]; */

UINT32 stream_address;

/* unsigned char ptr_head = 0; */

UINT32 ptr_tail = 0;

    PCIC80STAT ReturnVal;

UINT32 *ping_pong_addr = (UINT32 *) 0x010107D0;
UINT32 *changemod = (UINT32 *) 0x010107D4;

unsigned short local_alloc[AX*AY/6];

unsigned char comp_table[5] = {10, 20, 40, 80, 100};

unsigned char last_comp = 2;

*****/

static void SignalHandler(UINT32 Signals);
static void init_alloc_table(UINT32 index);
static void init_alloc_table2(UINT32 index);

```

# NAWCWD TP 8442

```

void task (void *arg)
{
    unsigned int times=0;

    unsigned int dx,dy;
    int i,j;
    int lp;
    unsigned char vert_loop_index[6],horiz_loop_index[6];
    unsigned short jump_col[6];
    unsigned int jump_row[6];

    long *ptr;           /* temp pointer */
    PTREQ *p[14];        /* temp pointer to packet transfer structure */

    int temp_maxval;

    /* JCWtest */

    char string1[32];
    int temp_str;
    float temp_time;
    float wait_time;
    unsigned int junk_time;
    float frame_time;
    unsigned int frame_hit;

    int junki;
    unsigned short junkfill = 0;
    unsigned short * fillme = (unsigned short *) 0x90320000;

    int k,l;
    long templ;
    unsigned char tbuf;

    unsigned char * tbuf_pp0      = (unsigned char *) 0x010005fc;
    unsigned char * tbuf_pp1      = (unsigned char *) 0x010015fc;
    unsigned char * tbuf_pp2      = (unsigned char *) 0x010025fc;
    unsigned char * tbuf_pp3      = (unsigned char *) 0x010035fc;

    unsigned char min_first;

    unsigned char ppnexttask[4];
    unsigned short ppallocc[4];
    unsigned char current_block;
    unsigned short table_size[4];
    unsigned char pprequesting;
    unsigned char ppinfo[4];
    unsigned char ppdone;

    unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

    unsigned char DONE1;

    unsigned int temp_ulong;

    unsigned int first_dec;

    NOCACHE_INT(number_pixels[0]) = 0x80;

    NOCACHE_INT(global_mean) = 0x5e;

```

# NAWCWD TP 8442

```
*pp_stop_encode = 0;
```

```
/* initialize FIRST variable on all PPs - used for first pass # of bitplanes to
process */
```

```
tbuf_pp0[0] = 4;
tbuf_pp1[0] = 4;
tbuf_pp2[0] = 4;
tbuf_pp3[0] = 4;
```

```
VolSetDisplay(VOL_VGA8_640);          /* setup colour display for VIM-8 */
VolSetGreyLUT();
```

```
#ifdef HOST
```

```
ReturnVal = CilSigHandler(SignalHandler);
if (ReturnVal != CIL_OK)
```

```
{
```

```
    /** Cannot register signal handler ***/
```

```
while(1);
```

```
    return;
```

```
}
```

```
#endif
```

```
p[0] = (PTREQ *) (MP_PARM_RAM + 0x300);
p[1] = p[0] + 1;
p[2] = p[0] + 2;
p[3] = p[0] + 3;
p[4] = p[0] + 4;
p[5] = p[0] + 5;
p[6] = p[0] + 6;
p[7] = p[0] + 7;
p[8] = p[0] + 8;
p[9] = p[0] + 9;
p[10] = p[0] + 10;
p[11] = p[0] + 11;
p[12] = p[0] + 12;
p[13] = p[0] + 13;
```

```
/* Set MP list pointer to point to first PT */
ptr = (long *) (MP_PTREQ_PTR);
*ptr = (long) p[0];
```

```
/* DRAM (8 bit data) -> VRAM (raw image) */
```

p[2]->link = p[2];	/* point to next PT	*/
p[2]->word[0] = 0x80000000;	/* linear to VRAM	*/
p[2]->word[1] = 0x80300000;	/* Src address is DRAM	*/
p[2]->word[2] = 0xb4000240;	/* Dst address is VRAM	*/
p[2]->word[3] = 0x00038000;	/* Src B count Src A count	*/
p[2]->word[4] = 0x00ff0200;	/* Dst B count Dst A count	*/
p[2]->word[5] = 0x00;	/* Src C count	*/
p[2]->word[6] = 0;	/* Dst C count	*/
p[2]->word[7] = 0x8000;	/* Src B pitch	*/
p[2]->word[8] = 0x500;	/* Dst B pitch	*/
p[2]->word[9] = 0x00;	/* Src C pitch	*/
p[2]->word[10] = 0x0000;	/* Dst C pitch	*/
p[2]->word[11] = 0;	/* Src transparency upper	*/
p[2]->word[12] = 0;	/* Src transparency lower	*/
p[2]->word[13] = 0;	/* Reserved	*/
p[2]->word[14] = 0;	/* Reserved	*/

# NAWCWD TP 8442

```

/* DRAM (columns and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst <- this can be done 8 times */

p[4]->link = p[4];          /* point to next PT */
p[4]->word[0] = 0x80000002;  /* Contig. Mem to int w/dst updt */
p[4]->word[1] = 0x80320000;  /* Src address is DRAM */
p[4]->word[2] = 0x00000000;  /* Dst address is internal */
p[4]->word[3] = 0x00ff0010;  /* Src B count Src A count */
p[4]->word[4] = 0x00001000;  /* Dst B count Dst A count */
p[4]->word[5] = 0x00;       /* Src C count */
p[4]->word[6] = 0;          /* Dst C count */
p[4]->word[7] = 0x0400;     /* Src B pitch */
p[4]->word[8] = 0x000;      /* Dst B pitch */
p[4]->word[9] = 0x10;       /* Src C pitch */
p[4]->word[10] = 0x1000;    /* Dst C pitch */
p[4]->word[11] = 0;         /* Src transparency upper */
p[4]->word[12] = 0;         /* Src transparency lower */
p[4]->word[13] = 0;         /* Reserved */
p[4]->word[14] = 0;         /* Reserved */

/* internal RAM -> DRAM (columns and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[5]->link = p[5];          /* point to next PT */
p[5]->word[0] = 0x80000200;  /* int to Contig. Mem w/src updt */
p[5]->word[1] = 0x00000000;  /* Src address is internal */
p[5]->word[2] = 0x80320000;  /* Dst address is DRAM */
p[5]->word[3] = 0x00001000;  /* Src B count Src A count */
p[5]->word[4] = 0x00ff0010;  /* Dst B count Dst A count */
p[5]->word[5] = 0x00;       /* Src C count */
p[5]->word[6] = 0;          /* Dst C count */
p[5]->word[7] = 0x0000;     /* Src B pitch */
p[5]->word[8] = 0x400;      /* Dst B pitch */
p[5]->word[9] = 0x1000;     /* Src C pitch */
p[5]->word[10] = 0x0010;    /* Dst C pitch */
p[5]->word[11] = 0;         /* Src transparency upper */
p[5]->word[12] = 0;         /* Src transparency lower */
p[5]->word[13] = 0;         /* Reserved */
p[5]->word[14] = 0;         /* Reserved */

/* DRAM (rows and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst then this can be repeated 8 times */

p[6]->link = p[6];          /* point to next PT */
p[6]->word[0] = 0x80000002;  /* Contig. Mem to int w/dst updt */
p[6]->word[1] = 0x80320000;  /* Src address is DRAM */
p[6]->word[2] = 0x00000000;  /* Dst address is internal */
p[6]->word[3] = 0x00001000;  /* Src B count Src A count */
p[6]->word[4] = 0x00001000;  /* Dst B count Dst A count */
p[6]->word[5] = 0x00;       /* Src C count */
p[6]->word[6] = 0;          /* Dst C count */
p[6]->word[7] = 0x0000;     /* Src B pitch */
p[6]->word[8] = 0x000;      /* Dst B pitch */
p[6]->word[9] = 0x0000;     /* Src C pitch */
p[6]->word[10] = 0x1000;    /* Dst C pitch */
p[6]->word[11] = 0;         /* Src transparency upper */
p[6]->word[12] = 0;         /* Src transparency lower */
p[6]->word[13] = 0;         /* Reserved */
p[6]->word[14] = 0;         /* Reserved */

```

# NAWCWD TP 8442

```

/* internal RAM -> DRAM (rows and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[7]->link = p[7];          /* point to next PT          */
p[7]->word[0] = 0x80000200;  /* int to DRAM w/src updt   */
p[7]->word[1] = 0x00000000;  /* Src address is internal   */
p[7]->word[2] = 0x80320000;  /* Dst address is DRAM      */
p[7]->word[3] = 0x00001000;  /* Src B count Src A count   */
p[7]->word[4] = 0x00001000;  /* Dst B count Dst A count   */
p[7]->word[5] = 0x00;        /* Src C count               */
p[7]->word[6] = 0;           /* Dst C count               */
p[7]->word[7] = 0x0000;      /* Src B pitch               */
p[7]->word[8] = 0x000;       /* Dst B pitch               */
p[7]->word[9] = 0x1000;      /* Src C pitch               */
p[7]->word[10] = 0x0000;     /* Dst C pitch               */
p[7]->word[11] = 0;          /* Src transparency upper    */
p[7]->word[12] = 0;          /* Src transparency lower    */
p[7]->word[13] = 0;          /* Reserved                  */
p[7]->word[14] = 0;          /* Reserved                  */

/* internal RAM -> VRAM (rows and 8 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[8]->link = p[8];          /* point to next PT          */
p[8]->word[0] = 0x80000202;  /* int to VRAM w/dst updt   */
p[8]->word[1] = 0x00000000;  /* Src address is internal   */
p[8]->word[2] = 0xb4000240;  /* Dst address is VRAM      */
p[8]->word[3] = 0x00000800;  /* Src B count Src A count   */
p[8]->word[4] = 0x00030200;  /* Dst B count Dst A count   */
p[8]->word[5] = 0x00;        /* Src C count               */
p[8]->word[6] = 0;           /* Dst C count               */
p[8]->word[7] = 0x0000;      /* Src B pitch               */
p[8]->word[8] = 0x280;       /* Dst B pitch               */
p[8]->word[9] = 0x1000;      /* Src C pitch               */
p[8]->word[10] = 0x1000;     /* Dst C pitch               */
p[8]->word[11] = 0;          /* Src transparency upper    */
p[8]->word[12] = 0;          /* Src transparency lower    */
p[8]->word[13] = 0;          /* Reserved                  */
p[8]->word[14] = 0;          /* Reserved                  */

/* internal RAM -> VRAM (rows and 8 bit data) */
/* Can be used 1 times then change src and dst */

p[9]->link = p[9];          /* point to next PT          */
p[9]->word[0] = 0x80000000;  /* int to VRAM w/dst updt   */
p[9]->word[1] = 0x00000000;  /* Src address is internal   */
p[9]->word[2] = 0xb4000240;  /* Dst address is VRAM      */
p[9]->word[3] = 0x00000800;  /* Src B count Src A count   */
p[9]->word[4] = 0x00030200;  /* Dst B count Dst A count   */
p[9]->word[5] = 0x00;        /* Src C count               */
p[9]->word[6] = 0;           /* Dst C count               */
p[9]->word[7] = 0x0000;      /* Src B pitch               */
p[9]->word[8] = 0x500;       /* Dst B pitch               */
p[9]->word[9] = 0x1000;      /* Src C pitch               */
p[9]->word[10] = 0x0008;     /* Dst C pitch               */
p[9]->word[11] = 0;          /* Src transparency upper    */
p[9]->word[12] = 0;          /* Src transparency lower    */
p[9]->word[13] = 0;          /* Reserved                  */
p[9]->word[14] = 0;          /* Reserved                  */

```

# NAWCWD TP 8442

```

/* internal -> DRAM coefficient blocks (16 bit data)*/

    p[10]->link = p[10];          /* point to next PT */
    p[10]->word[0] = 0x80000000;   /* Contig. Mem to int w/dst updt */
    p[10]->word[1] = 0x00008000;   /* Src address is DRAM */
    p[10]->word[2] = 0x80320000;   /* Dst address is internal */
    p[10]->word[3] = 0x00000400;   /* Src B count Src A count */
    p[10]->word[4] = 0x000f0040;   /* Dst B count Dst A count */
    p[10]->word[5] = 0x00;         /* Src C count */
    p[10]->word[6] = 0;           /* Dst C count */
    p[10]->word[7] = 0x000;       /* Src B pitch */
    p[10]->word[8] = 0x0400;      /* Dst B pitch */
    p[10]->word[9] = 0x1000;      /* Src C pitch */
    p[10]->word[10] = 0x40;       /* Dst C pitch */
    p[10]->word[11] = 0;          /* Src transparency upper */
    p[10]->word[12] = 0;          /* Src transparency lower */
    p[10]->word[13] = 0;          /* Reserved */
    p[10]->word[14] = 0;          /* Reserved */

/* DRAM -> internal RAM (byte stream 8 bit data) */

    p[12]->link = p[12];          /* point to next PT */
    p[12]->word[0] = 0x80000200;   /* SDRAM to internal w/src updt */
    p[12]->word[1] = 0x80380000;   /* Src address is external */
    p[12]->word[2] = 0x01000630;   /* Dst address is internal PRAM */
    p[12]->word[3] = 0;           /* Src B count Src A count */
    p[12]->word[4] = 0;           /* Dst B count Dst A count */
    p[12]->word[5] = 0x00;        /* Src C count */
    p[12]->word[6] = 0;           /* Dst C count */
    p[12]->word[7] = 0x0000;      /* Src B pitch */
    p[12]->word[8] = 0x000;       /* Dst B pitch */
    p[12]->word[9] = 0;           /* Src C pitch */
    p[12]->word[10] = 0x1000;     /* Dst C pitch */
    p[12]->word[11] = 0;          /* Src transparency upper */
    p[12]->word[12] = 0;          /* Src transparency lower */
    p[12]->word[13] = 0;          /* Reserved */
    p[12]->word[14] = 0;          /* Reserved */

/* IE = disable() & 0xfbf0fffe; */

/* IE = 0x01; */

/* INTPEN = 0xF0000; */

vert_loop_index[0] = 16;
vert_loop_index[1] = 4;
vert_loop_index[2] = 1;
vert_loop_index[3] = 1;
vert_loop_index[4] = 1;

horiz_loop_index[0] = 15;
horiz_loop_index[1] = 5;
horiz_loop_index[2] = 1;
horiz_loop_index[3] = 1;
horiz_loop_index[4] = 1;

jump_col[0] = 16;
jump_col[1] = 64;
jump_col[2] = 256;
jump_col[3] = 256;
jump_col[4] = 256;

```



# NAWCWD TP 8442

```

jump_row[0] = 0x1000; /* not used */
jump_row[1] = 0x3000;
jump_row[2] = 0xF000;
jump_row[3] = 0x10000;
jump_row[4] = 0x10000;

init_alloc_table(2);

/*****
*/

/* Setup Initial graphics on output screen */

/* DRAM (8 bit data) -> VRAM (raw image) */

    p[2]->link = p[2];          /* point to next PT          */
    p[2]->word[0] = 0x80000000;  /* linear to VRAM          */
    p[2]->word[1] = 0x80300000;  /* Src address is DRAM      */
    p[2]->word[2] = 0xb4000440;  /* Dst address is VRAM     */
    p[2]->word[3] = 0x00007800;  /* Src B count Src A count 64x480 */
size image */
    p[2]->word[4] = 0x01df0040;  /* Dst B count Dst A count  */
    p[2]->word[5] = 0x00;        /* Src C count              */
    p[2]->word[6] = 0;           /* Dst C count              */
    p[2]->word[7] = 0x00;        /* Src B pitch              */
    p[2]->word[8] = 0x280;       /* Dst B pitch              */
    p[2]->word[9] = 0x00;        /* Src C pitch              */
    p[2]->word[10] = 0x0000;     /* Dst C pitch              */
    p[2]->word[11] = 0;          /* Src transparency upper   */
    p[2]->word[12] = 0;          /* Src transparency lower   */
    p[2]->word[13] = 0;          /* Reserved                  */
    p[2]->word[14] = 0;          /* Reserved                  */

p[2]->word[1] = (long)&logo;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from external to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

```

# NAWCWD TP 8442

```

p[2]->word[2] = 0xb4000200;          /* Dst address is VRAM          */
p[2]->word[3] = 0x00000940;          /* Src B count Src A count      64x37 size
image */
p[2]->word[4] = 0x00240040;          /* Dst B count Dst A count      */

p[2]->word[1] = (long)&dsp_util;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from external to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

p[2]->word[2] = 0xb400ae80;          /* Dst address is VRAM          */
p[2]->word[3] = 0x00000940;          /* Src B count Src A count      64x37 size
image */
p[2]->word[4] = 0x00240040;          /* Dst B count Dst A count      */

p[2]->word[1] = (long)&frame_rate;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from external to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

/* DRAM (8 bit data) -> VRAM (raw image) */

p[2]->link = p[2];                  /* point to next PT            */
p[2]->word[0] = 0x80000000;          /* linear to VRAM              */
p[2]->word[1] = 0x80300000;          /* Src address is DRAM          */
p[2]->word[2] = 0xb4000240;          /* Dst address is VRAM          */
p[2]->word[3] = 0x00038000;          /* Src B count Src A count      */
p[2]->word[4] = 0x00ff0200;          /* Dst B count Dst A count      */
p[2]->word[5] = 0x00;                /* Src C count                  */
p[2]->word[6] = 0;                  /* Dst C count                  */
p[2]->word[7] = 0x8000;              /* Src B pitch                  */
p[2]->word[8] = 0x800;              /* Dst B pitch                  */
p[2]->word[9] = 0x00;              /* Src C pitch                  */
p[2]->word[10] = 0x0000;             /* Dst C pitch                  */
p[2]->word[11] = 0;                 /* Src transparency upper       */
p[2]->word[12] = 0;                 /* Src transparency lower       */
p[2]->word[13] = 0;                 /* Reserved                     */
p[2]->word[14] = 0;                 /* Reserved                     */

```

# NAWCWD TP 8442

```

#ifdef HOST
/*
** Tell host we are ready to start decoding
*/
ReturnVal = CilRaiseSignalNumber(C80ReadyBit);
if (ReturnVal != CIL_OK)
{
    /*** Cannot raise signal ***/
while(1);
    return;
}

#endif

while (1)
{

/* TCOUNT = 0xffffffff; */

    if (TCOUNT ==0)
        TCOUNT = 0xffffffff;

p[9]->word[2] = 0xb4000240;          /* Dst address is VRAM          */

p[7]->word[3]  = 0x00001000;
p[7]->word[4]  = 0x00001000;
p[7]->word[6]  = 0x00000000;
p[7]->word[8]  = 0x00000000;
p[7]->word[10] = 0x00001000;

/*
** Wait for request from host
*/

time_in_wait = TCOUNT;

    CilReadMailbox(0, (PUINT32)ping_pong_addr);

    while( *ping_pong_addr == ptr_tail)
    {
        for (i=0;i<1000;i++)
        {
            i = i + 1;
            i = i - 1;
        }
        CilReadMailbox(0, (PUINT32)ping_pong_addr);
    }

    stream_address = 0x90013000 + (ptr_tail * 0x4000);

    ptr_tail = ptr_tail + 1;
    if (ptr_tail == d_buffers) ptr_tail = 0;

```

```

/* Do time calculations */

time_in_wait = time_in_wait - TCOUNT;

frame_time = (frame_hit - TCOUNT)*0.000025;
frame_hit = TCOUNT;

decode_time = TCOUNT;

/* Read dynamic values from the bitstream */

temp_ulong = NOCACHE_INT(* (int *) stream_address);

first_dec = ((temp_ulong & 0xff000000) >> 24);

    tbuf_pp0[0] = first_dec;
    tbuf_pp1[0] = first_dec;
    tbuf_pp2[0] = first_dec;
    tbuf_pp3[0] = first_dec;

global_mean = ((temp_ulong & 0x00ff0000) >> 16);

global_maxval = ((temp_ulong & 0x0000ff00) >> 8);

*changemod = (temp_ulong & 0x000000ff);

#ifdef variable_compression
/* Check for change in compression */
temp_ulong = (NOCACHE_INT(* (int *) (stream_address + 4))) & 0x000000ff;

    if (temp_ulong != last_comp)
    {
        init_alloc_table(temp_ulong);
        last_comp = temp_ulong;
    }

#else

/* Check for change in compression */
temp_ulong = (NOCACHE_INT(* (int *) (stream_address + 4)));

    if (temp_ulong < 480) temp_ulong = 480;
    if (temp_ulong > 16640) temp_ulong = 16640;

    if (temp_ulong != last_comp)
    {
        init_alloc_table2(temp_ulong);
        last_comp = temp_ulong;
    }
#endif

```

# NAWCWD TP 8442

```

/*****
/***** Decode Section *****/
/*****

/* Do Bit stream conversion to coefficients */

/* 128 byte blocks make up 32x32 coefficient blocks that are organized */
/* as 8x16 */

#ifdef DECODE_STREAM

ppnexttask[0] = 0;
ppnexttask[1] = 0;
ppnexttask[2] = 0;
ppnexttask[3] = 0;

current_block = 0;
ppdone = 0;

p[12]->word[1] = stream_address + 8;          /* Read Src address from queued
requests */

while((INTPEN & 0xF0000) != 0xF0000);    /* Wait for all PPs to catch up */

*pp_stop_encode = 0;

command(0x0000200F);          /* send msg interrupt to all PPs */

while (ppdone!=4)
{
    if ((INTPEN & 0xF0000) != 0x00)
    {

        DONE1 = 0;

        /* find out which PP requested service */

        for (pprequesting=0;pprequesting<4;pprequesting++)
        {
            if ((INTPEN & (1<<(16+pprequesting))) != 0) break;
        }

        /* clear the interrupt flag */

        INTPEN = 0x10000<<pprequesting;

        /* Get third pair of coefficients from PP and send in next block of bits to
        decode if not complete */
        if ((ppnexttask[pprequesting] == 3) && (!DONE1))
        {

            p[10]->word[1] = 0x00008000 + (pprequesting<<12);          /* Src
            address is internal RAM2 */
            p[10]->word[2] = 0x80348000 + (ppinfo[pprequesting]>>3)*0x04000 +
            (ppinfo[pprequesting]&7)*64;          /* Dst for block changes */

            *ptr = (long) p[10];

            /* kick off 1024 byte (32x16 words) coefficient block transfer from
            internal to SDRAM */
            PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

```

# NAWCWD TP 8442

```

        p[10]->word[1] = p[10]->word[1] + 0x400;          /* Src for block
changes is internal RAM2 */
        p[10]->word[2] = p[10]->word[2] + 0x200;          /* Dst address is
SDRAM */

        /* kick off 1024 byte (32x16 words) coefficient block transfer from
internal to SDRAM */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        ppnexttask[pprequesting] = 1;

        if (current_block != 40)
        {
            /* read in TBYTES number of bytes from the bit stream for this block to
the appropriate PP */

            table_size[pprequesting] = local_alloc[current_block];

            ppinfo[pprequesting] = current_block;
            ppnexttask[pprequesting] = 1;
            current_block = current_block + 1;

            *(unsigned short *) (0x01000600 + (pprequesting<<12)) =
table_size[pprequesting] - *changemod;

            p[12]->word[2] = 0x01000630 + (pprequesting<<12);          /* Dst
address is internal Parameter RAM */

            *ptr = (long) p[12];

            /* Read the number of bytes that need to be transferred */
            p[12]->word[3] = p[12]->word[4] = p[12]->word[9] =
table_size[pprequesting];

            /* write out TBYTES value for this block to the appropriate PP */
            *(unsigned short *) (0x01000600 + (pprequesting<<12)) =
table_size[pprequesting] - *changemod;

            /* kick off transfer from SDRAM to internal */
            PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ &
0x02);

            command(0x00002000 + (1<<pprequesting));          /* send msg
interrupt to PP that requested service */

        }
        else ppdone = ppdone + 1;

        DONE1 = 1;

    }

    /* Get first or second pair of coefficients from PP */
    if ((ppnexttask[pprequesting] > 0) && (ppnexttask[pprequesting] < 3)) &&
(!DONE1))
    {
        p[10]->word[1] = 0x00008000 + (pprequesting<<12);          /* Src
address is internal RAM2 */
        p[10]->word[2] = 0x80320000 + (ppinfo[pprequesting]>>3)*0x04000 +
(ppinfo[pprequesting]&7)*64 + 0x14000*(ppnexttask[pprequesting]-1);          /*
Dst for block changes */
    }

```

# NAWCWD TP 8442

```

    *ptr = (long) p[10];

    /* kick off 2048 byte coefficient block transfer from internal to SDRAM
    */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    p[10]->word[1] = p[10]->word[1] + 0x400;          /* Src for block
changes is internal RAM2 */
    p[10]->word[2] = p[10]->word[2] + 0x200;          /* Dst address is
SDRAM */

    /* kick off 1024 byte (32x16 words) coefficient block transfer from
internal to SDRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

    command(0x00002000 + (1<<pprequesting));          /* send msg
interrupt to PP that requested service */

    ppnexttask[pprequesting] += 1;

    DONE1 = 1;

    }

    /* Give PP a block of bits to decode */
    /* Move bitstream from SDRAM to internal */

    if ((ppnexttask[pprequesting] == 0) && (!DONE1))
    {

        /* read in TBYTES number of bytes from the bit stream for this block to the
        appropriate PP */

        table_size[pprequesting] = local_alloc[current_block];

        ppinfo[pprequesting] = current_block;
        ppnexttask[pprequesting] = 1;
        current_block = current_block + 1;

        p[12]->word[2] = 0x01000630 + (pprequesting<<12);          /* Dst address
is internal Parameter RAM */

        *ptr = (long) p[12];

        /* Read the number of bytes that need to be transferred */
        p[12]->word[3] = p[12]->word[4] = p[12]->word[9] =
table_size[pprequesting];

        /* write out TBYTES value for this block to the appropriate PP */
        *(unsigned short *) (0x01000600 + (pprequesting<<12)) =
table_size[pprequesting] - *changemod;

        /* kick off transfer from SDRAM to internal */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        command(0x00002000 + (1<<pprequesting));          /* send msg interrupt
to PP that requested service */

        DONE1 = 1;

    }

```

```

    }
}

*pp_stop_encode = 1;
command(0x0000200F);          /* send msg interrupt to all PPs */

#endif

/* FIRST variable is sent in with the bitstream */

/* Determine new FIRST variable common to all PPs */

    /* select minimum FIRST value */
/* min_first = tbuf_pp0[1];

    if (tbuf_pp1[1] < min_first) min_first = tbuf_pp1[1];
    if (tbuf_pp2[1] < min_first) min_first = tbuf_pp2[1];
    if (tbuf_pp3[1] < min_first) min_first = tbuf_pp3[1];
*/
/* copy it to all PPs */

/* tbuf_pp0[0] = min_first;
   tbuf_pp1[0] = min_first;
   tbuf_pp2[0] = min_first;
   tbuf_pp3[0] = min_first;
*/

CilWriteMailbox(1, ptr_tail);

#ifdef DECODE

for (lp=4; lp>=0; lp--)
{

/* Do 32 rows */

for (i=0; i<horiz_loop_index[lp]; i++)
{

    if (i==0)
    {
        p[6]->word[1] = 0x80320000;          /* Src address is SDRAM          */
        p[7]->word[2] = 0x80320000;          /* Dst address is SDRAM          */
    }

    p[6]->word[2] = 0x00000000;          /* Dst address is internal      */
    p[7]->word[1] = 0x00000000;          /* Src address is internal      */

    *ptr = (long) p[6];

    if (i==0)
    {
        switch (lp) {
            case 0:
                {
                    p[6]->word[3] = 0x00001000;
                    p[6]->word[4] = 0x00001000;
                    p[6]->word[5] = 0x00000000;
                    p[6]->word[7] = 0x00000000;
                    p[6]->word[9] = 0x00000000;
                }
            }
        }
    }
}

```



# NAWCWD TP 8442

```

p[7]->word[3] = 0x00001000;
p[7]->word[4] = 0x00001000;
p[7]->word[6] = 0x00000000;
p[7]->word[8] = 0x00000000;
p[7]->word[10] = 0x00000000;
break;
}
case 1:
{
p[6]->word[3] = 0x00ff0002;
p[6]->word[4] = 0x00000c00;
p[6]->word[5] = 0x00000005;
p[6]->word[7] = 0x00000004;
p[6]->word[9] = 0x00000800;

p[7]->word[3] = 0x00000c00;
p[7]->word[4] = 0x00ff0002;
p[7]->word[6] = 0x00000005;
p[7]->word[8] = 0x00000004;
p[7]->word[10] = 0x00000800;
break;
}
case 2:
{
p[6]->word[3] = 0x007f0002;
p[6]->word[4] = 0x00000f00;
p[6]->word[5] = 0x0000000e;
p[6]->word[7] = 0x00000008;
p[6]->word[9] = 0x00001000;

p[7]->word[3] = 0x00000f00;
p[7]->word[4] = 0x007f0002;
p[7]->word[6] = 0x0000000e;
p[7]->word[8] = 0x00000008;
p[7]->word[10] = 0x00001000;
break;
}
case 3:
{
p[6]->word[3] = 0x003f0002;
p[6]->word[4] = 0x00000400;
p[6]->word[5] = 0x00000007;
p[6]->word[7] = 0x00000010;
p[6]->word[9] = 0x00002000;

p[7]->word[3] = 0x00000400;
p[7]->word[4] = 0x003f0002;
p[7]->word[6] = 0x00000007;
p[7]->word[8] = 0x00000010;
p[7]->word[10] = 0x00002000;
break;
}
case 4:
{
p[6]->word[3] = 0x001f0002;
p[6]->word[4] = 0x00000100;
p[6]->word[5] = 0x00000003;
p[6]->word[7] = 0x00000020;
p[6]->word[9] = 0x00004000;

p[7]->word[3] = 0x00000100;
p[7]->word[4] = 0x001f0002;

```

# NAWCWD TP 8442

```

        p[7]->word[6] = 0x00000003;
        p[7]->word[8] = 0x00000020;
        p[7]->word[10] = 0x00004000;
        break;
    }
    default:
        break;
}
}

/* kick off transfer from SDRAM to internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002001);          /* send msg interrupt to PP0 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

/* kick off transfer from SDRAM to internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002002);          /* send msg interrupt to PP1 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

/* kick off transfer from SDRAM to internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002004);          /* send msg interrupt to PP2 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

/* kick off transfer from SDRAM to internal */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002008);          /* send msg interrupt to PP3 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x10000)==0x00); /* poll PP0 */
    INTPEN = 0x10000;                /* clear the interrupt flag */
#endif

    if (lp==0)
        *ptr = (long) p[9];
    else
        *ptr = (long) p[7];

p[9]->word[1] = 0x00000000;          /* Dst address is VRAM */

/* start xfer from internal to DRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

```

# NAWCWD TP 8442

```

if (lp==0)
{
    p[9]->word[2] = p[9]->word[2] + 0x280;
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
    p[9]->word[2] = p[9]->word[2] - 0x280 + 5120;
    p[9]->word[1] = p[9]->word[1] + 0x1000;
}

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x20000)==0x00);    /* poll PP1 */
    INTPEN = 0x20000;                  /* clear the interrupt flag */
#endif

    /* start xfer from internal to DRAH */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

if (lp==0)
{
    p[9]->word[2] = p[9]->word[2] + 0x280;
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
    p[9]->word[2] = p[9]->word[2] - 0x280 + 5120;
    p[9]->word[1] = p[9]->word[1] + 0x1000;
}

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x40000)==0x00);    /* poll PP2 */
    INTPEN = 0x40000;                  /* clear the interrupt flag */
#endif

    /* start xfer from internal to DRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

if (lp==0)
{
    p[9]->word[2] = p[9]->word[2] + 0x280;
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
    p[9]->word[2] = p[9]->word[2] - 0x280 + 5120;
    p[9]->word[1] = p[9]->word[1] + 0x1000;
}

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x80000)==0x00);    /* poll PP3 */
    INTPEN = 0x80000;                  /* clear the interrupt flag */
#endif

    /* start xfer from internal to DRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

if (lp==0)
{
    p[9]->word[2] = p[9]->word[2] + 0x280;
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
    p[9]->word[2] = p[9]->word[2] - 0x280 + 5120;
    p[9]->word[1] = p[9]->word[1] + 0x1000;
}

```

# NAWCWD TP 8442

```

p[7]->word[2] = p[7]->word[2] + jump_row[lp];

} /* end for i */

/* Do the appropriate number of columns depending on the scale */

if (lp!=0)
    for (i=0; i<vert_loop_index[lp-1]; i++)
    {

        if (i==0)
        {
            p[4]->word[1] = 0x80320000;          /* Src address is DRAM          */
            p[5]->word[2] = 0x80320000;          /* Dst address is DRAM          */
        }

        p[4]->word[2] = 0x00000000;          /* Dst address is internal      */

        *ptr = (long) p[4];

#ifdef DEBUG
        /* JCWtest */
        if (i==1)
        {
            for (junki=0;junki<256*256;junki=junki+1)
            {
                fillme[junki] = junkfill;
                junkfill = junkfill + 1;
            }
        }
#endif

        if (i==0)
        {
            switch (lp) {
                case 1:
                {
                    p[4]->word[3] = 0x00ef0010;
                    p[4]->word[4] = 0x00000f00;
                    p[4]->word[5] = 0x00000000;
                    p[4]->word[7] = 0x00000400;
                    p[4]->word[9] = 0x00000000;

                    p[5]->word[3] = 0x00000f00;
                    p[5]->word[4] = 0x00ef0010;
                    p[5]->word[6] = 0x00000000;
                    p[5]->word[8] = 0x00000400;
                    p[5]->word[10] = 0x00000000;
                    break;
                }
                case 2:
                {
                    p[4]->word[3] = 0x000f0002;
                    p[4]->word[4] = 0x00000f00;
                    p[4]->word[5] = 0x00000077;
                    p[4]->word[7] = 0x00000004;
                    p[4]->word[9] = 0x00000800;

                    p[5]->word[3] = 0x00000f00;
                    p[5]->word[4] = 0x000f0002;
                }
            }
        }
    }

```

# NAWCWD TP 8442

```

    p[5]->word[6] = 0x00000077;
    p[5]->word[8] = 0x00000004;
    p[5]->word[10] = 0x00000800;
    break;
}
case 3:
{
    p[4]->word[3] = 0x001f0002;
    p[4]->word[4] = 0x00000f00;
    p[4]->word[5] = 0x0000003b;
    p[4]->word[7] = 0x00000008;
    p[4]->word[9] = 0x00001000;

    p[5]->word[3] = 0x00000f00;
    p[5]->word[4] = 0x001f0002;
    p[5]->word[6] = 0x0000003b;
    p[5]->word[8] = 0x00000008;
    p[5]->word[10] = 0x00001000;
    break;
}
case 4:
{
    p[4]->word[3] = 0x000f0002;
    p[4]->word[4] = 0x000003c0;
    p[4]->word[5] = 0x0000001d;
    p[4]->word[7] = 0x00000010;
    p[4]->word[9] = 0x00002000;

    p[5]->word[3] = 0x000003c0;
    p[5]->word[4] = 0x000f0002;
    p[5]->word[6] = 0x0000001d;
    p[5]->word[8] = 0x00000010;
    p[5]->word[10] = 0x00002000;
    break;
}
default:
    break;
}
}

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002001);          /* send msg interrupt to PP0      */
#endif

if ((i==0)&(lp==1))
    time_colin = 0xffffffff - TCOUNT;

p[4]->word[1] = p[4]->word[1] + jump_col[lp-1];          /* Src address is DRAM
*/

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002002);          /* send msg interrupt to PP1      */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp-1]; /* Src address is DRAM
*/

```

# NAWCWD TP 8442

```

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002004);          /* send msg interrupt to PP2      */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp-1]; /* Src address is DRAM
*/

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002008);          /* send msg interrupt to PP3      */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp-1]; /* Src address is DRAM
*/

p[5]->word[1] = 0x00000000;          /* Src address is internal      */

*ptr = (long) p[5];

#ifdef DEBUG
    while((INTPEN & 0x10000)==0x00); /* poll PP0 */
    INTPEN = 0x10000;                /* clear the interrupt flag */
#endif

/* kick off transfer from internal to DRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

if (i==0)
    time_colout = 0xffffffff - TCOUNT;

p[5]->word[2] = p[5]->word[2] + jump_col[lp-1]; /* Dst address is DRAM
*/

#ifdef DEBUG
    while((INTPEN & 0x20000)==0x00); /* poll PP1 */
    INTPEN = 0x20000;                /* clear the interrupt flag */
#endif

/* kick off transfer from internal to DRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp-1]; /* Dst address is DRAM
*/

#ifdef DEBUG
    while((INTPEN & 0x40000)==0x00); /* poll PP2 */
    INTPEN = 0x40000;                /* clear the interrupt flag */
#endif

/* kick off transfer from internal to DRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp-1]; /* Dst address is DRAM
*/

```

# NAWCWD TP 8442

```

#ifndef DEBUG
    while((INTPEN & 0x80000)==0x00); /* poll PP3 */
    INTPEN = 0x80000; /* clear the interrupt flag */
#endif

/* kick off transfer from internal to DRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

p[5]->word[2] = p[5]->word[2] + jump_col[lp-1]; /* Dst address is DRAM
*/

    } /* end for i */

} /* end for lp */

last_dtime = decode_time;
decode_time = decode_time - TCOUNT;

times = junk_time;
junk_time = TCOUNT;

/* DRAM (8 bit data) -> VRAM (raw image) */

    p[2]->link = p[2]; /* point to next PT */
    p[2]->word[0] = 0x80000000; /* linear to VRAM */
    p[2]->word[1] = 0x80300000; /* Src address is DRAM */
    p[2]->word[2] = 0xb4000240; /* Dst address is VRAM */
    p[2]->word[3] = 0x00110010; /* Src B count Src A count */
    p[2]->word[4] = 0x00110010; /* Dst B count Dst A count */
    p[2]->word[5] = 0x00; /* Src C count */
    p[2]->word[6] = 0; /* Dst C count */
    p[2]->word[7] = 0xb0; /* Src B pitch */
    p[2]->word[8] = 0x280; /* Dst B pitch */
    p[2]->word[9] = 0x00; /* Src C pitch */
    p[2]->word[10] = 0x0000; /* Dst C pitch */
    p[2]->word[11] = 0; /* Src transparency upper */
    p[2]->word[12] = 0; /* Src transparency lower */
    p[2]->word[13] = 0; /* Reserved */
    p[2]->word[14] = 0; /* Reserved */

/* times = times + 1;
if (times == 10) times = 0; */

#ifdef SHOW_WAITFORHOST
/* sprintf(string1,"%d",times); */
sprintf(string1,"%f",(times*0.00002));

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
    else
    {
        temp_str = 160;
    }

    p[2]->word[1] = (long)&number_pixels;
    p[2]->word[1] += temp_str;
}

```

```

p[2]->word[2] = 0xb4042b00 + i*16;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);
}
#endif

#ifdef SHOW_DECODETIME
/*temp_time = 1.0/(decode_time*0.00000002); */
temp_time = (decode_time*0.00002); /* time in milliseconds */

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
if (string1[i]!='.')
{
temp_str = string1[i];
temp_str = temp_str - 48;
temp_str = temp_str * 16;
}
else
{
temp_str = 160;
}

p[2]->word[1] = (long)&number_pixels;
p[2]->word[1] += temp_str;
p[2]->word[2] = 0xb4045580 + i*16;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);
}

#endif

#ifdef SHOW_DSPUTIL
/*temp_time = 1.0/(decode_time*0.00000002); */
temp_time = (decode_time*100.0)/((decode_time + time_in_wait)*1.0); /*
percentage of time spend processing */

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
if (string1[i]!='.')
{
temp_str = string1[i];
temp_str = temp_str - 48;
temp_str = temp_str * 16;

```



```

    }
else
{
    temp_str = 160;
}

p[2]->word[1] = (long)&number_pixels;
p[2]->word[1] += temp_str;
p[2]->word[2] = 0xb4005e80 + i*16;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);
}

#endif

#ifdef SHOW_FRAMERATE
sprintf(string1,"%f",1000.0/frame_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
else
{
    temp_str = 160;
}

p[2]->word[1] = (long)&number_pixels;
p[2]->word[1] += temp_str;
p[2]->word[2] = 0xb4010b00 + i*16;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);
}

#endif

#ifdef SHOW_FRAMETIME
sprintf(string1,"%f",frame_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {

```

```

    temp_str = string1[i];
    temp_str = temp_str - 48;
    temp_str = temp_str * 16;
}
else
{
    temp_str = 160;
}

p[2]->word[1] = (long)&number_pixels;
p[2]->word[1] += temp_str;
p[2]->word[2] = 0xb4048000 + i*16;

*ptr = (long) p[2];

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);
}

#endif

/* DRAM (8 bit data) -> VRAM (raw image) */

p[2]->link = p[2]; /* point to next PT */
p[2]->word[0] = 0x80000000; /* linear to VRAM */
p[2]->word[1] = 0x80300000; /* Src address is DRAM */
p[2]->word[2] = 0xb4000240; /* Dst address is VRAM */
p[2]->word[3] = 0x00038000; /* Src B count Src A count */
p[2]->word[4] = 0x00ff0200; /* Dst B count Dst A count */
p[2]->word[5] = 0x00; /* Src C count */
p[2]->word[6] = 0; /* Dst C count */
p[2]->word[7] = 0x8000; /* Src B pitch */
p[2]->word[8] = 0x800; /* Dst B pitch */
p[2]->word[9] = 0x00; /* Src C pitch */
p[2]->word[10] = 0x0000; /* Dst C pitch */
p[2]->word[11] = 0; /* Src transparency upper */
p[2]->word[12] = 0; /* Src transparency lower */
p[2]->word[13] = 0; /* Reserved */
p[2]->word[14] = 0; /* Reserved */

command(0x0000200F); /* send msg interrupt to PP1 */

#endif

/* ptr_tail = ptr_tail + 1;
if (ptr_tail == d_buffers) ptr_tail = 0; */

#ifdef dumb_HOST
/*
** Tell host this frame is completed decoding
*/
RetVal = CilRaiseSignalNumber(C80DoneBit);
if (RetVal != CIL_OK)
{
    /** Cannot raise signal ***/
while(1);
return;
}

```

```

    }

#endif

junk_time = junk_time - TCOUNT;

    } /* end while */

} /* end task */

extern int ep_runpp0;

main()
{
    int i;
    unsigned int temp;
    unsigned int *src_ptr = (unsigned int *)0x90018000;
    unsigned int *dst_ptr = (unsigned int *)0x80000000;

    /* REFCNTL = 0xffff0138; */ /* setup up dram and sdram to correct refresh rate
    for 40 Mhz C80*/
    REFCNTL = 0xffff0186; /* setup up dram and sdram to correct refresh rate for 50
    Mhz C80*/

    command(0xc000000f);          /* reset and halt PP0,1,2,3          */

    *(int *)0x010001b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010011b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010021b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010031b8 = (int)&ep_runpp0; /* initialize task vector */

    /* upload PP code */

    /* memcpy( 0x80000000, 0x90200000, 0x9020bae0 - 0x90200000); */

    for (i=0;i<20000;i++)
    {
        temp = *src_ptr++;
        *dst_ptr++ = temp;
    }

    for (i=0;i<20000;i++)
    {
        temp = temp + 1;
        temp = temp - 1;
    }

    command(0x3000000f);          /* start PP0,1,2,3 by unhalting it */
                                /* all will take its task interrupt*/

    /* Basic init functions */
#ifdef HOST
    InterruptInit();             /* Init ME interrupts */
#endif

    PtReqInit();                 /* Init the ME PT functions */
    TaskInitTasking();           /* Init tasking */
    IclInstallPtdMalloc();       /* Install protected malloc and free function to ME */
    IclPTInit(15);               /* Init the Icl PT server task with a priority of 15 */
}

```

# NAWCWD TP 8442

```

#ifdef HOST
/*
** Initialise the Cil
** Declare 4 buffers of 256 bytes each.
** These buffers are not used here - choose minimum sizes.
*/
CilInit(4,256);
#endif

TaskResume(TaskCreate(-1,task, NULL, 14, 4096)); /* Start task */

while(1==1); /* loop */
}

/*****
*
* Function : SignalHandler
* Args      : UINT32 Signals
*
* Description:
* Signals      Signals raised by host
*
* SignalHandler will be called when host raises signal
*
* Return Values:
* None
*
*****/

void
SignalHandler(UINT32 Signals)
{
    if ((Signals & HostRequestBitMask) != 0)
    {
        /*
        ** Read where shared data can be found
        */

        ReturnVal = CilReadMailbox(0, (PUINT32)ping_pong_addr);
        /* ReturnVal = CilReadMailbox(1, (PUINT32)changemod); */

        /* buf_addr[ptr_head] = *ping_pong_addr; */

        if (ReturnVal != CIL_OK)
        {
            /*** Cannot read from mailbox ***/
            while(1);
        }

        /* ptr_head = ptr_head + 1;
        if (ptr_head == d_buffers) ptr_head = 0; */

    }
}

/* Initialize the adaptive bit allocation tables on the PPs */

void
init_alloc_table(UINT32 index)
{
    int t_alloc;

```

```

int p_alloc;
int r_alloc;
int k,lp;

t_alloc = YSIZE*FSIZE/comp_table[index] - headersize;
p_alloc = t_alloc/((AY*AX)/6);
r_alloc = t_alloc%((AY*AX)/6);

for(k=0;k<((AY*AX)/6);k++)
    local_alloc[k] = p_alloc;

lp = 0;
while (r_alloc>0)
{
    local_alloc[lp++] ++;
    r_alloc--;
}

}

void
init_alloc_table2(UINT32 index)
{
    int t_alloc;
    int p_alloc;
    int r_alloc;
    int k,lp;

    t_alloc = index - headersize;
    p_alloc = t_alloc/((AY*AX)/6);
    r_alloc = t_alloc%((AY*AX)/6);

    for(k=0;k<((AY*AX)/6);k++)
        local_alloc[k] = p_alloc;

    lp = 0;
    while (r_alloc>0)
    {
        local_alloc[lp++] ++;
        r_alloc--;
    }

}

```

# NAWCWD TP 8442

```

/*****
*   Filename : pcic80.cmd
*
*   Description:
*
*   PCI/C80 linker file
*
*****/
-c
-heap 0x100000
-stack 0x100000
-l mp_cio.lib
-l \pcic80\lib\mp_task.lib
-l mp_int.lib
-l mp_rts.lib
-l mp_ptreq.lib
-l mp_ppcmd.lib
-l ppcmd.lib
-l \pcic80\lib\icl.lib
-l \pcic80\lib\vol.lib
-l \pcic80\lib\bgl.lib
-l \pcic80\lib\vil24.lib
-l \pcic80\lib\cil.lib

MEMORY
{
    RAM0 : o=0x00000000    l = 0x00800
    RAM1 : o=0x00000800    l = 0x00800
    RAM2 : o=0x00008000    l = 0x00800
    RESERV : o=0x01000000    l = 0x00200
    MPPRAM : o=0x010007D8    l = 0x00028
    SDRAM : o=0x80000000    l = 0x300000
    DRAM : o=0x90000000    l = 0x18000
    DRAM2 : o=0x90018000    l = 0x1e8000
    UNINIT : o=0x90200000    l = 0x200000
    IMAGE : o=0x80300000    l = 0x80000
    SPOT : o=0x80380000    l = 0x10000
    VRAM_PAL : o=0xB0000000    l = 0x200000
    VRAM_VGA : o=0xB4000000    l = 0x400000
}

SECTIONS
{
    /*
    * The following section must be defined for all program that
    * use the CIL. The section must appear in the first 8Mb
    * of DRAM and must be long enough to include all buffers
    * plus 128 bytes. This example is big enough for 4*256byte
    * buffers.
    *
    * See the user guide for more information.
    */
    .lsidram : {
        _CildRAMBase = .;
        . += 0x600;
    } > DRAM

    .text : > DRAM
    .cinit : > DRAM
    .const : > DRAM
    .switch : > DRAM
    .data : > DRAM

```

## NAWCWD TP 8442

```
.bss:    > DRAM
.cio:    > DRAM
.pcinit  :    > DRAM

.ptext   :    load > DRAM2, run SDRAM

font:    load > DRAM2, run SDRAM

.systemem :    > UNINIT
.stack    :    > UNINIT
sh_vars   :    > MPPRAM

rawimage:  > IMAGE
stream:    > SPOT
}

/*****
* End of file pcic80.cmd
*****/
```

# NAWCWD TP 8442

```

/*****
*   Filename : pcic80a.cmd
*
*   Description:
*
*   PCI/C80 linker file
*
*****/
-pc
-l d:\mvp\src\newlib\pp_rts.lib

-pstack 256

MEMORY
{
    RAM0 : o=0x00000000   l = 0x00800
    RAM1 : o=0x00000800   l = 0x00800
    RAM2 : o=0x00000800   l = 0x00800
    RESERV : o=0x01000000   l = 0x00200
    PRAM : o=0x01000200   l = 0x00600
    SDRAM : o=0x80000000   l = 0x800000
    DRAM : o=0x90400000   l = 0x400000
    VRAM_PAL : o=0xB0000000   l = 0x200000
    VRAM_VGA : o=0xB4000000   l = 0x400000
}

SECTIONS
{
    .text : > DRAM
    .ptext : > DRAM
    .cinit : > DRAM
    .const : > DRAM
    .switch : > DRAM
    .data : > DRAM
    .bss : > DRAM
    .cio : > DRAM
    .sysmem : > DRAM
    .stack : > DRAM

    .pcinit : > DRAM

    .pbss : (PASS) > PRAM
    .psysmem : (PASS) > PRAM
    .pstack : (PASS) > PRAM
}

```



# NAWCWD TP 8442

```

/*****
*
*   Function   : $quick_syms
*   Args      : none
*   Passed in :
*
*   Description:
*
*       $quick_syms will be called to calculate the number of
*       symbols needed for this pass thru the refinement pass.
*
*   Return Values:
*       None
*
*****/

stats_flag .seta0
save_array .seta1
kids_list  .seta8
stats_flag_base .seta9
i          .setx0
index      .setx9
std        .setx2

tempd1     .setd1
tempd2     .setd2
tempd3     .setd3
tempd4     .setd4

.global $quick_syms
.global $stats_flag
.global $save_array
.global $save_index
.global $syms_to_do
.global $read_more
.global $left_off
.global $link_list
.global $total_links

.align 128

$quick_syms:

    stats_flag_base =uw *(pba + $stats_flag)
    save_array =uw *(pba + $save_array)
    kids_list = &*(pba + $link_list)
    std = 0

    tempd4 = 64

    tempd2 =uh *(pba + $left_off)
    tempd1 = 256 - tempd2

    tempd3 =ub *(pba + $read_more)
    tempd3 = tempd3 - 0
    tempd2 =[eq.z] tempd4
    tempd1 =[eq] 192

    i = tempd2

```

```

le0 = endquick
lrs0 = tempd1 - 1
*(pba + $syms_to_do) =uh a15
*(pba + $save_index) =uh a15

tempd1 =ub *(pba + $total_links)

le1 = incr
lrs1 = tempd1 - 1
index = 0
tempd2 = 251

tempd1 =ub *(kids_list + index)
tempd3 = tempd1<<8
stats_flag = stats_flag_base + tempd3
index = index + 1

tempd1 =ub *(stats_flag + i)
a15 = tempd1 & 5
tempd1 =[ne.z] tempd1 & tempd2
br =[ne] incr
*(stats_flag + i) =ub tempd1
tempd1 = i + tempd3

*(save_array +[std]) =uh tempd1

a15 = std - 240
lc0 =[gt] 0
std = std + 1

incr:
    nop

endquick:
    i = i + 1

    *(pba + $syms_to_do) =uh std

    *(pba + $left_off) =uh i
    tempd1 = 0

    a15 = i - 256
    tempd1 =[ne] 1

end_quick_syms:
    br = iprs
    *(pba + $read_more) =ub tempd1
    nop

```

## INTERFRAME ENCODING

```

;Written by Jim Witham, 3-27-97
; addconv.s
; To replace the following C Code that just took too much time...
;   for(i=0;i<2048;i++)
;   {
;       img[i] = img[i] + oldmean
;
;       pic[i] = img[i];
;       if (img[i]>255)
;           pic[i] = 255;
;       if (img[i]<0)
;           pic[i] = 0;
;   }
;

.global $add_n_conv8

$add_n_conv8:
    d1 = d1<<3
    d6 =rh d1                ;d1 is mean that is passed in
                             ;replicate it to both words

    sr = 0xad
    x0 = 1
    x1 = 2

    d0 = 0
    d4 = 0x08000800          ; 0800>>3 = 0100
    d5 = 0x07f807f8          ; 07f8>>3 = 00ff
    a2 = &*(dba + 1)

    le2 = L48
    lrs2 = 1023

    a1 = &*(dba)
    a0 = &*(dba)

    d1 = *(a1++=[x0])
    d1 =me d1 + d6
    d3 = (d0 & @mf) | (d1 & ~@mf) ;if (val-mean<0) then d3 = 0 else d3 =
d1
    d2 =me d1 - d4
    d3 = (d3 & @mf) | (d5 & ~@mf) ;if (val-mean>0x800) then d3 = 0x7f8
else d3 = d3

    d3 = d3 >>u 3            ; shift down to proper scaling
    *(a2++=[x1]) =b d3
    || d2 =h1 d3

L48:
    *(a0++=[x1]) =b d2

    sr = 0x36

br = iprs
    nop
    nop

```

## NAWCWD TP 8442

```
/* DECLARATIONS FOR ARITHMETIC CODING AND DECODING */

/* Size of arithmetic code values */

#define Code_value_bits 9 /* Number of bits in a code value */
typedef short code_value; /* Type of arithmetic code value */

#define Top_value (((long)1<<Code_value_bits)-1) /* Largest code val */

/* Half and Quarter points in code value range */

#define First_qtr (Top_value/4+1) /* Points after first quarter */
#define Half      (2*First_qtr)   /* Points after first half */
#define Third_qtr (3*First_qtr)   /* Points after third quarter */
```

```

/*****
**
** bitebits.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $new_dbits_wo
** $new_dbits_nwo
** $new_dbits2
** $do_syms
** $encode_symbol
** I_DIV_JW
** $update_model
** $bit_plus_follow
** $new_output_bit
**
*****/
.global $new_dbits_wo
.global $new_dbits_nwo

.global $new_dbits2

.global $output_bit
.global $do_syms

.global $T_BYTES
.global $byte_stream
.global $STOP

.global $bit_index

.global $ztl      ;signed  short pointer sh *(xba + $ztl)
.global $THRESH   ;signed  int   sw *(xba + $THRESH)
.global $stats_flag ;unsigned char pointer *(xba + $stats_flag)
.global $char_to_index ;unsigned char pointer &*(xba
$char_to_index) +
.global $stats_val ;signed  short pointer *(xba + $stats_val)
.global $TMASK     ;unsigned short uh *(xba + $TMASK)
.global $BITE      ;signed  int   sw *(xba + $BITE)
.global $SHFT_DN   ;signed  int   sw *(xba + $SHFT_DN)

.global $update_model
.global $encode_symbol

.global $new_output_bit

.global $time_cdwo
.global $time_cdnwo
.global $time_cd2wo
.global $time_sym

.global $list
.global $list_index

.global $pruned_children

.global $getaway_address
.global $quick_getaway

.global $qstats_val

```

# NAWCWD TP 8442

```

.global $qindex
.global $qcount

tempd1      .setd0
stats_flag  .setd1
stats_val   .setd2
tempd2      .setd3
tflg        .setd4
tmask       .setd5
tempd3      .setd6
t           .setd7
sym         .seta12

maxsyms     .set63

.align      512

/*****
*
*   Function   : $new_dbits_wo
*   Args      : m ,c1,c3,s
*   Passed in : (d1,d2,d3,d4)
*
*   Description:
*       m - the current index of this coefficient
*
*       c1 - the relative index of the first child
*
*       c3 - the relative index of the third child
*
*       s - the subblock that we are in
*
*   $new_dbits_wo will be called to scan a coefficient that is
*   on the second or third tier of the tree. The coefficient
*   has already been checked previously for being significant
*   on a previous bit-plane, now it is check against the
*   current bit-plane threshold for significance, or pass
*   down.
*
*   Return Values:
*       None
*
*****/

$new_dbits_wo:

    d0 = &(sp --= 28)
    *(sp + 16) =w iprs

    *(sp + 12) =w a4
||    *(sp + 8) =w d6

    *(sp + 4) =w a12
||    *(sp + 0) =w d7

    *(sp + 24) = d4                ;save s onto stack

    d5 = d1

    a1 =uw *(xba + $stats_val)      ;*(a1) = stats_val[s*256]
    a2 =uw *(xba + $stats_flag)
    a3 =uw *(xba + $zt1)            ;*(a3 + [x0]) = zt1[]

```

# NAWCWD TP 8442

```

    d6 = d4 << 9          ; 256 elements each 2 bytes long times s (<< 9 = *
512)
    a1 = a1 + d6

    d6 = d4 << 8          ; 256 elements each 1 byte long times s (<< 8 = *
256)
    a2 = a2 + d6

    d6 = d4 << 7          ; ztl[s*64] (where ztl[] is a short)
    a3 = a3 + d6

    d5 = d5 + (d4<<8)      ; m = m + s*256

    *(sp + 20) =uh d5

    a0 = a2                ;*(a0)          = stats_flag[s*256]

    x0 = d1                ;x0 = m
    x1 = d3                ;*(a2 + [x1]) = stats_flag[c3]
    x2 = d3 + 1            ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

    a2 = a2 + d2            ;*(a2)          = stats_flag[s*256 + c1]
                           ;*(a2 + [1])    = stats_flag[c2 = c1 + 1]

    ;*(a1 + [x0]) = stats_val[s*256 + m]
    ;*(a0 + [x0]) = stats_flag[s*256 + m]
    ;*(a2)          = stats_flag[s*256 + c1]
    ;*(a2 + [1])    = stats_flag[s*256 + c2] (c2 = c1 + 1)
    ;*(a2 + [x1])   = stats_flag[s*256 + c3]
    ;*(a2 + [x2])   = stats_flag[s*256 + c4] (c4 = c3 + 1)

    tmask =uh *(xba + $TMASK)

    tflg =sh *(a3 + [x0])    ;fetch ztl[m + s*64]
    a15 = tflg&tmask
    tflg = 1 || tflg =[ne] a15    ;calculate tflg

    tflg = tflg - 0
    br =[z] no_change
    stats_val =sh *(a1 + [x0])    ;fetch stats_val[m]
    tempd3 = |stats_val|          ;take abs(stats_val[m])

;      tempd1 =sw *(xba + $time_cdwo)
;      tempd1 = tempd1 + 1
;      *(xba + $time_cdwo) =w tempd1

    tempd1 = tflg << 1
    a4 = tflg

    call = mod_flags
    d4 = *(sp + 24)            ;reload s from stack into d4
    tempd2 =ub *(a2)

    tflg = a4

no_change:
    t = tempd3 & tmask          ;form TMASK&abs(stats_val[m])
    tempd1 =sw *(xba + $SHFT_DN) ;fetch SHFT_DN
    tempd1 = -tempd1
    t = t >>u -tempd1          ;form (TMASK&abs(stats_val[m]))>>SHFT_DN

```

```

;compute sym

    tempd1 =sw *(xba + $BITE)
    tempd1 = %tempd1
    t = t - 0
    tempd1 =[eq] a15
    stats_val = stats_val - 0
    tempd1 =[le] a15

finish_up:
    tempd2 = (~tflg)&1
    tempd2 = tempd2 + t
    tempd2 = tempd2 + tempd1

    x1 = tempd2

    t = t - 0
    br =[le] L68
    nop
    nop

;if (t>0)
;    stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)/(1<<(BITE-1)))+THRESH/(2<<(BITE-1))));
/* or */
;    stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)>>(BITE-1))+(THRESH>>BITE));

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

    tflg = 1 - tempd1

    tmask =u tempd2 * t
    tempd1 = - tempd1

    a1 = *(pba + $qstats_val)
    a2 = *(pba + $qindex)
    x2 =ub *(pba + $qcount)

    tempd3 = tempd3 - (tmask >>u -tflg)
    tempd3 = tempd3 - (tempd2 >>u -tempd1)

    *(a1 + [x2]) =h tempd3
    tempd1 =uh *(sp + 20)
    *(a2 + [x2]) =h tempd1
    tempd1 = x2 + 1
    *(pba + $qcount) =ub tempd1

L68:
    tempd1 = 0
    t = t - 0
    tempd1 =[nz] 4 ;calculate (t!=0)<<2
    *(a0 + [x0]) =ub tempd1

    a15 = tempd1 - 0
    br =[eq] nosig
    x0 =uh *(pba + $list_index)
    a15 = x0 - 254
    br =[ge] nosig
    tempd1 =uh *(sp + 20)

```



```

a0 =uw *(pba + $list)
tempd2 = x0 + 1
*(a0 + [x0]) =uh tempd1
*(pba + $list_index) =uh tempd2

nosig:
a0 = &*(pba + $sym_array)
x0 =uh *(pba + $sym_index)

nop

tempd2 = x0 + 1
*(pba + $sym_index) =uh tempd2

;sym_array[sym_index++] = sym;

*(a0 + [x0]) =ub x1

; if (sym_index>63) do_syms();

tempd1 = tempd2 - maxsyms

call =[gt] $do_syms
nop
nop

done_more:
a4 = *(sp + 12)
a12 = *(sp + 4)
br = *(sp + 16)

d6 =sw *(sp + 8)
|| d7 =sw *(sp + 0)
d0 = &*(sp += 28)

/*****
*
* Function : $new_dbits_nwo
* Args : m ,c1,c3,s
* Passed in :(d1,d2,d3,d4)
*
* Description:
* m - the current index of this coefficient
*
* c1 - the relative index of the first child
*
* c3 - the relative index of the third child
*
* s - the subblock that we are in
*
* $new_dbits_nwo will be called when the current coefficient
* and all of its children have been deemed insignificant,
* from the pass down flag.
*
* Return Values:
* None
*
*****/

```

```
$new_dbits_nwo:
```

```

    a2 =uw *(xba + $stats_flag)
    d5 = d4 << 8
    a2 = a2 + d5

;   a0 = a2                      ;*(a0 + [x0])      = stats_flag[s*256 + m]

    x0 = d1                      ;x0 = m
    x1 = d3                      ;*(a2 + [x1]) = stats_flag[s*256 + c3]
    x2 = d3 + 1                  ;*(a2 + [x2]) = stats_flag[c4 = c3 + 1]

    a2 = a2 + d2                  ;*(a2)          = stats_flag[s*256 + c1]
                                ;*(a2 + [1])    = stats_flag[c2 = c1 + 1]

;   tempd2 =ub *(a0 + [x0])

;   tempd2 = tempd2 & 252
;   *(a0 + [x0]) =ub tempd2

;       tempd1 =sw *(xba + $time_cdnwo)
;       tempd1 = tempd1 + 1
;       *(xba + $time_cdnwo) =w tempd1

    tempd1 = 2
    tempd2 =ub *(a2)

```

```
mod_flags:
```

```

    tempd2 = tempd2 | tempd1
    *(a2) =ub tempd2

    tempd2 =ub *(a2 + [1])
    tempd2 = tempd2 | tempd1
    *(a2 + [1]) =ub tempd2

    tempd2 =ub *(a2 + [x1])
    tempd2 = tempd2 | tempd1
    *(a2 + [x1]) =ub tempd2

    tempd2 =ub *(a2 + [x2])

    tempd2 = tempd2 | tempd1
    *(a2 + [x2]) =ub tempd2

    a2 = &*(pba + $pruned_children)
    x2 = d4      ;s is in d4
    nop
    tempd1 =ub *(a2 + x2)

    br = iprs

    tempd1 = tempd1 + 1
    *(a2 + x2) =ub tempd1

    .align 512

```

```

/*****
*
*   Function   : $new_dbits2
*   Args      : m,s
*   Passed in : d1,d2
*
*   Description:
*       $new_dbits2 will be called to scan the fourth tier in the
*       zero tree.
*
*   Return Values:
*       None
*
*****/

```

```
$new_dbits2:
```

```

    d0 = &*(sp --= 24)
    *(sp + 16) =w iprs

    *(sp + 8) =w a4
||    *(sp + 4) =w d6

    *(sp + 0) =w d7

a0 =uw *(xba + $stats_flag)
a1 =uw *(xba + $stats_val)      ;*(a1) = stats_val[s*256]

d5 = d1

d3 = d2 << 9
a1 = a1 + d3

d3 = d2 << 8
a0 = a0 + d3

d5 = d5 + (d2 << 8)

*(sp + 20) =uh d5                ;index = m + s*256

x0 = d1                        ;x0 = m

;*(a0 + [x0]) = stats_flag[s*768 + m]
;*(a1 + [x0]) = stats_val[s*768 + m]

tmask =uh *(xba + $TMASK)

stats_val =sh *(a1 + [x0])      ;fetch stats_val[m]
tempd3 = |stats_val|           ;take abs(stats_val[m])
t = tempd3 & tmask             ;form TMASK&abs(stats_val[m])

;    tempd1 =sw *(xba + $time_cd2wo)
;    tempd1 = tempd1 + 1
;    *(xba + $time_cd2wo) =w tempd1

tempd1 =sw *(xba + $SHFT_DN)    ;fetch SHFT_DN
tempd1 = -tempd1
t = t >>u -tempd1              ;form (TMASK&abs(stats_val[m]))>>SHFT_DN

```

```

;compute sym

    tempd2 =sw *(xba + $BITE)
    tempd2 = %tempd2

    stats_val = stats_val - 0
    tempd2 =[le] a15

    t = t - 0
    tempd2 =g[eq] a15

    tempd2 =[ne] tempd2 + 1          ;form (t!=0) and add to (1<<BITE)-1

    t = t - 0
    br =[le] L69

    x1 = tempd2 + t          ;sym = t + (((t!=0)&&(stats_val[m]>0)) ?
    ((1<<BITE)-1):0) + (t!=0)
    a4 = &*(xba + $char_to_index)

;if (t>0)
;    stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)/(1<<(BITE-
1))+THRESH/(2<<(BITE-1)))));
; /* or */
; stats_val[m] = (abs(stats_val[m]) - ((t*THRESH)>>(BITE-1)) + (THRESH>>BITE));

    tempd1 =sw *(xba + $BITE)
    tempd2 =uh *(xba + $THRESH)

    tflg = 1 - tempd1

    tmask =u tempd2 * t
    tempd1 = - tempd1

    a1 = *(pba + $qstats_val)
    a2 = *(pba + $qindex)
    x2 =ub *(pba + $qcount)

    tempd3 = tempd3 - (tmask >>u -tflg)
    tempd3 = tempd3 - (tempd2 >>u -tempd1)

    *(a1 + [x2]) =h tempd3
    tempd1 =uh *(sp + 20)
    *(a2 + [x2]) =h tempd1
    tempd1 = x2 + 1
    *(pba + $qcount) =ub tempd1

L69:
    tempd1 = 0
    t = t - 0
    tempd1 =[nz] 4          ;calculate (t!=0)<<2
    *(a0 + [x0]) =ub tempd1

    a15 = tempd1 - 0
    br =[eq] nosig2
    x0 =uh *(pba + $list_index)
    a15 = x0 - 254
    br =[ge] nosig2
    tempd1 =uh *(sp + 20)
    a0 =uw *(pba + $list)
    tempd2 = x0 + 1
    *(a0 + [x0]) =uh tempd1

```

# NAWCWD TP 8442

```

    *(pba + $list_index) =uh tempd2
nosig2:
    a0 = &*(pba + $sym_array)
    x0 =uh *(pba + $sym_index)

    tempd2 = x0 + 1
    *(pba + $sym_index) =uh tempd2
;sym_array[sym_index++] = sym;

    *(a0 +[x0]) =ub x1
; if (sym_index>63) do_syms();

    tempd1 = tempd2 - maxsyms

    call =[gt] $do_syms
    nop
    nop

    a4 =sw  *(sp + 8)

    br = *(sp + 16)

    d6 =sw  *(sp + 4)
    || d7 =sw  *(sp + 0)
    d0 = &*(sp += 24)

;

```

```

.global $index_to_char
.global $No_of_symbols
.global $bits_to_follow

.global $high
; .global I_DIV
.global $freq
.global $low
.global $cum_freq

.global $bit_plus_follow

.global $sym_array
.global $sym_index

.align 512

/*****
*
* Function : do_syms
* Args      : none
*
* Description:
*   do_syms will be called to empty the symbol cache.
*
* Return Values:
*   None
*
*****/
$do_syms:
    d0 = &(sp --= 20)
    *(sp + 16) =w iprs

    *(sp + 12) =w a4
||    *(sp + 8) =w d6

    *(sp + 4) =w a12
||    *(sp + 0) =w d7

    a12 = &(xba + $char_to_index)
    a4 = &(xba + $sym_array)

    d6 =uh *(xba + $sym_index)
    x8 =ub *a4++

more_syms:
    call = $encode_symbol
    d7 =ub *(a12 + x8)
    d1 = d7

    call = $update_model
nop
    d1 = d7

;    d1 =sw *(xba + $time_sym)
;    d1 = d1 + 1
;    *(xba + $time_sym) =w d1

```

```

        d1 =sw  *(xba + $STOP)
        d1 = d1 - 1
        br =[eq] get_out
        x8 =ub  *a4++
        d6 = d6 - 1

        br =[gt] more_syms
get_out:
        a0 = *(pba + $qstats_val)
        a1 = *(pba + $qindex)

        d1 =uh  *(xba + $sym_index)
        a4 = &*(xba + $sym_array)
|| d6 = d1 - d6
        a2 = *(pba + $stats_val)

do_over:
        d1 =ub  *a4++
        d1 = d1 - 1
        br =[le] too_small
        nop
        nop

        x0 =h  *a1++
        d1 =h  *a0++
        *(a2 + [x0]) =h d1

too_small:
        d6 = d6 - 1
        br =[gt] do_over
        nop
        nop

        *(pba + $qcount) =ub a15

get_out2:

        a4 =w  *(sp + 12)
|| d6 =w  *(sp + 8)

        a12 =w *(sp + 4)
|| d7 =w  *(sp + 0)

        d3 = *(sp + 16)
        d0 = d3
        || d2 = *(pba + $getaway_address)
        d1 =ub *(pba + $quick_getaway)
        d1 = d1 - 1
        d0 =[eq] d2
        d1 =sw  *(xba + $STOP)
        d1 = d1 - 0
        d0 =[eq] d3

; br = d0
br = *(sp + 16)
*(pba + $sym_index) =uh a15
d0 = &*(sp += 20)

```

```

/*****
*
*   Function : encode_symbol
*   Args      : none
*
*   Description:
*       encode_symbol will be called to perform the arithmetic
*       encoding of a symbol. This routine was lifted from a C
*       compiled program and included here, for cache coherency.
*
*   Return Values:
*       None
*
*****/

```

```

Sencode_symbol:
    x0 = d1
    a0 = &*(xba + $cum_freq)
    d3 =sw  *(xba + $low)
    d2 =sw  *(xba + $high)
    d1 = x0 << 1

    d4 = d2 - d3
||    d2 =g  a0

    a1 = d1 + d2
||    d0 = &*(sp --= 4)

    d4 = d4 + 1
||    *(sp) =w  iprs

    d1 =uh1  d4
||    d5 =sh  *(a1 - 2)

    d1 =u d5 * d1
||    d2 =uh1  d5
    d2 =u d2 * d4

    d1 =u d5 * d4
||    d2 = d2 + d1
    call = I_DIV_JW

    d1 = d1 + (d2 << 16)
||    x1 =sh  *a0
    d2 = x1

    d0 =uh1  d4
||    d1 =sh  *(a0 + [x0])

    d0 =u d0 * d1
||    d2 =uh1  d1
    d2 =u d4 * d2

    d1 =u d4 * d1
||    d2 = d0 + d2
    d1 = d1 + (d2 << 16)

    d4 = d5 + d3
||    d2 = x1
    call = I_DIV_JW
    d4 = d4 - 1
    *(xba + $high) =w  d4

```



# NAWCWD TP 8442

```

    d2 = d5 + d3
    d1 = d4 - (1 \ \ 8)
    br =[ge] L25
    *(xba + $low) =w d2
    d1 =[ge] d2 - (1 \ \ 8)

L19:
    call = $bit_plus_follow
    nop
    d1 = 0

    br = L23
    d1 =sw *(xba + $high)
    d1 = d1 << 1

L20:
    d1 =sh *(xba + $bits_to_follow)

    d2 = d1 + 1
    || d3 =sw *(xba + $low)
    *(xba + $bits_to_follow) =h d2

    d2 = d3 - (1 \ \ 7)
    || d1 =sw *(xba + $high)
    br = L22

    d1 = d1 - (1 \ \ 7)
    || *(xba + $low) =w d2
    *(xba + $high) =w d1

L21:
    call = $bit_plus_follow
    nop
    d1 = 1

    d1 =sw *(xba + $low)

    d1 = d1 - (1 \ \ 8)
    || d2 =sw *(xba + $high)

    d1 = d2 - (1 \ \ 8)
    || *(xba + $low) =w d1
    *(xba + $high) =w d1

L22:
    d1 = d1 << 1

L23:

    d4 = d1 + 1
    || d1 =sw *(xba + $low)
    d2 = d1 << 1
    d1 = d4 - (1 \ \ 8)
    br =[lt] L19
    *(xba + $high) =w d4
    *(xba + $low) =w d2

L24:
    d1 = d2 - (1 \ \ 8)

L25:
    br =[ge] L21
    nop
    d1 =[lt] d2 - (1 \ \ 7)

```

```

        br =[lt] L30
nop
        d1 =[ge] d4 - 384

        br =[lt] L20
        br =[ge] L30
nop

nop

L30:
        br = *(sp)
        nop
        d0 = &*(sp += 4)

```

```

/*****
*
*   Function : I_DIV_JW
*   Args      : none
*
*   Description:
*       I_DIV_JW will be called to perform an Integer Divide.
*
*   Return Values:
*       None
*
*****/

```

```

;*****
;* I_DIV.ASM   v1.10   - Integer Divide
;* Copyright (c) 1993-1995 Texas Instruments Incorporated
;*****

```

```

; +-----+
; |       i_div.asm = PP assembly program that is used to return a 32-bit |
; |       signed integer quotient from 32-bit signed integer             |
; |       division when called by a C program.                           |
; |                                                                       |
; +-----+

```

```

.global    I_DIV_JW

```

```

; +-----+
; | 32-bit Signed Integer Word Divide Subroutine :
; |   o Input 32-bit signed integer Operand 1 is in d1 (numerator).
; |   o Input 32-bit signed integer Operand 2 is in d2 (divisor).
; |   o Output 32-bit signed integer is in d5 (Answer = quotient).
; |   o Output 32-bit signed remainder is discarded.
; |   o 0 input divisor produces 0x80000000 output with overflow set.
; |   o Quotient = 0x80000000 sets overflow.
; |   o Number of Stack Words used = 3.
; |   o MF register is saved.
; |
; |   o NOTE: Loop Counter 2 Registers are used but NOT restored !
; +-----+

```

# NAWCWD TP 8442

```

; +-----+
; |      32 bit / 32 bit ==> 32 bit signed quotient      |
; |      Signed PP Integer Division                      |
; |      Numerator / Denominator = Quotient + Remainder (discarded) |
; |      Divide by 0 produces 80000000 and sets sr(V)      |
; |      Divide Overflow is not possible if Divisor is non-zero, |
; |      except 80000000/ffffffff = 80000000 will set sr(V). |
; | MF register is preserved.                               |
; +-----+

```

```

; .ptext      ; PP assembly code

```

```

arg1: .setd1 ; input argument 1 = Numerator (32 low bits)
arg2: .setd2 ; input argument 2 = Divisor (32 bits)
ans:.setd5 ; answer = 32 bit signed quotient
Div:.setd3 ; Input Divisor
Num:.setd4 ; Input high Numerator = 0
Tmp:.setd5 ; ALU output for each DIVI

```

```

; .align 8*16; start on a 16-instruction boundary

```

```

I_DIV_JW:      ; Signed Word Integer Divide: Ans = Op1 / Op2

```

```

Div = 0 - | arg2 |      ; negate | divisor |
|| *(sp-=|3|) = Div      ; || push Div
br = [z] Div_By_0        ; Divide By 0 ?
Num = 0                  ; high numerator = 0
|| *(sp+[1]) = mf        ; || push mf
|| *(sp+[2]) = Num       ; || push Num
mf = | arg1 |            ; input lo | numerator |

```

```

lrse2 = 29              ; loop count - 1
Tmp = divi(Div, Num=Num) ; 1-st divide iterate
    Tmp = divi(Div, Num=Tmp [n] Num ) ; 2-nd divide iterate
LoopSW: Tmp = divi(Div, Num=Tmp [n] Num ) ; divide iterate 3-32

```

```

ans = mf                ; | ans | = mf
|| Div = *sp++          ; || pop Div
Num = arg1 ^ arg2        ; quotient sign
|| br = iprs            ; || return
ans =[n] -ans           ; quotient is negative,
|| mf = *sp++           ; || pop mf
Num = *sp++              ; pop Num

```

```

Div_By_0:              ; Divide By 0 \_____ Optional Error
Div_Ovfl:              ; Divide Overflow /_____ Return Code

```

```

br = iprs              ; return
|| Div = *sp++         ; || pop Div
mf = *sp++              ; pop mf
ans = 0 - 1<<31        ; returns 0x80000000, sets sr(V)
|| Num = *sp++         ; || pop Num      ...[END]

```

```

.global $update_model

```

# NAWCWD TP 8442

```

/*****
*
*   Function : update_model
*   Args      : none
*
*   Description:
*       update_model will be called to update the arithmetic
*       model's parameters. This routine was lifted from a C
*       compiled program and included here, for cache coherency.
*
*   Return Values:
*       None
*
*****/

```

```

$update_model:
    a0 = &(xba + $cum_freq)
    nop

    a1 = d1
||    d1 =sh *a0
    d1 = d1 - 75
    br =[ne] L6
    nop
    a2 =g [ne.ncvz] a1

    d2 =sw *(xba + $No_of_symbols)
    d1 = d2 - 0
    br =[lt] L6
    nop
    a2 =g [lt.ncvz] a1

    d1 =g a0
    le1 = L5 - 8

    d5 = d2 << 1
||    d3 = &(xba + $freq)
    a0 = d5 + d1

    d4 = d2 << 1
||    lrs1 = d2
    a8 = d4 + d3
    d2 = 0

L4:
    d1 =sh *a8
    d1 = d1 + 1
    d3 = d1 >>u 31
    d1 = d1 + d3
    d1 = d1 >>s 1
    d1 =sh0 d1
    *a8 =h d1
    *a0-- =h d2
    d1 =sh *a8--
    d2 = d2 + d1

```

```

L5:      a2 =g  a1
L6:      d3 = &*(xba + $freq)
          d1 = a2 << 1
          d5 = a2 << 1
          d4 = d3 + d1
          a0 = d5 + d3
          a8 = d4 - 2
          nop

          d4 =sh  *a8
||      d3 =sh  *a0
          d3 = d3 - d4
          br =[ne] L9
          d2 =g [ne.ncvz] a2
          d3 =[ne] d2 - a1

L7:      d1 = d1 - 2
||      d3 =sh  *---a8

          a2 = a2 - 1
||      d4 =sh  *---a0
          d3 = d4 - d3
          br =[eq] L7
          nop
          nop

L8:      d2 =g  a2
          d3 = d2 - a1
L9:      br =[ge] L11
          nop
          nop

          x0 = &*(xba + $index_to_char)
          nop
          a8 =ub  *(a2 + x0)
          x8 = &*(xba + $char_to_index)
          a9 =ub  *(a1 + x0)
          *(a2 + x0) =b  a9
          *(a1 + x0) =b  a8
          *(a8 + x8) =b  a1
          *(a9 + x8) =b  a2

L11:     d3 =sh  *a0
          d3 = d3 + 1

          d3 = a2 - 0
||      *a0 =h  d3
          br =[le] L15
          br =[le] L16
          nop

          le2 = L14 - 8
          d2 = a2 - 1
          d3 = &*(xba + $cum_freq)
          lrs2 = d2
          a0 = d1 + d3

```

```

        nop

L13:      d1 =sh  *--a0
          d1 = d1 + 1
          *a0 =h  d1

L14:      br = L16
          nop
L15:      nop

L16:      br = iprs
          nop
          nop

.global $bit_plus_follow

/*****
*
*   Function : bit_plus_follow
*   Args      : none
*
*   Description:
*       bit_plus_follow will be called to output several bits that
*       have been encoded by the arithmetic encoder.
*
*   Return Values:
*       None
*
*****/

$bit_plus_follow:
    d0 = &*(sp --= 8)
    *(sp + 4) =w  iprs
    call = $new_output_bit
    nop

    d6 = d1
||      *(sp + 0) =w  d6

    d1 =sh  *(xba + $bits_to_follow)
    d1 = d1 - 0
    br =[le] L36
    nop
    d1 =[gt] d6 - 0

    d6 = 1 || d6 =[ne] a15
L34:     call = $new_output_bit
    nop
    d1 = d6

    d1 =sh  *(xba + $bits_to_follow)
    d1 = d1 - 1
    d1 =sh0  d1

    d1 = d1 - 0
||      *(xba + $bits_to_follow) =h  d1

```

# NAWCWD TP 8442

```

        br =[gt] L34
        nop
        nop

L36:
        br = *(sp + 4)
        nop

        d6 =sw  *(sp + 0)
||      d0 = &*(sp += 8)

        .global $bit_index

/*****
*
*   Function : new_output_bit
*   Args      : d1 - the bit to append
*
*   Description:
*       new_output_bit will be called to append a single bit to
*       the bitstream array.
*
*   Return Values:
*       None
*
*****/

$new_output_bit:

        d2 =sw  *(xba + $STOP)
        d2 = d2 - 1
        br =[eq] iprs
        d2 =uh  *(xba + $bit_index)
        d4 = d2 >>u 3
||      d3 =uw  *(xba + $byte_stream)

        a0 = d4 + d3
        d0 =uh  *(xba + $T_BYTES)

        d4 = (d2&7)
||      d3 =ub  *a0
        d3 = d3 | (d1 << d4)
        *a0 =b  d3
||      d5 = d2 + 1

        *(xba + $bit_index) =h  d5

        d5 = d5 - (d0 << 3)

        br =g iprs
        d1 = 1 || d1 =[lt] a15      ;calculate new STOP value
        *(xba + $STOP) =w  d1

end_do_syms2:
        nop

        .global end_do_syms2

```

NAWCWD TP 8442

```
mpcl -s -g -c -i\pcic80\include mp.c  
mvplnk -x mp.obj num2.obj pp0.out pcic80.cmd -o mp.out -m mp.map
```



## NAWCWD TP 8442

```
erase *.o
ppcl -s -k main.c
ppcl -s -k j_enc.c
ppcl -s -k modlp.c
ppcl -k -o2 codefast.c
ppasm bitebits.s
ppasm subpass3.s
ppasm hvtry.s
ppasm addconv.s
ppasm mean2.s
ppasm ztr.s
ppasm diffs.s
mvplnk -x main.o hvtry.o mean2.o addconv.o j_enc.o codefast.o ztr.o bitebits.o
subpass3.o modlp.o diffs.o pcic80a.cmd -t runpp0 -o pp0.out -m pp0.map
```

# NAWCWD TP 8442

```

/*****
**
** codenew.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains code_dlist2 subroutine.
**
*****/

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

/* #define NO_MULTI_BITPLANE */

#define XSIZE 512
#define YSIZE 256
#define NSCALES 5
#define AX 16 /* = XSIZE/BS */
#define AY 8 /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

#define maxsyms 63

/* REAL-TIME: 2 Zerotrees/partition, last subband truncated */

extern unsigned char emubrk;

extern unsigned int *stomp_flag;

extern unsigned char *stats_flag;

extern int STOP;

extern unsigned short syms_to_do;

extern unsigned short *save_array;
extern unsigned char read_more;
extern unsigned short left_off;

extern int FIRST_DEC;

extern unsigned char RB_DEC;

extern unsigned char PASS_DEC;

extern unsigned short THRESH;

extern unsigned short *list;
extern unsigned short list_index;

extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short *stats_val;

extern unsigned short symbols_read;

extern int BITE;

extern unsigned char PASS_ENC;
extern unsigned short TMASK;

```

```

extern int SHFT_DN, RB_ENC;

extern unsigned char sym_array[256];
extern unsigned short sym_index;

extern short *qstats_val;
extern unsigned short *qindex;

extern unsigned char qcount;

extern unsigned char pruned_children[8]; /* used to keep count of the number of
pruned */
/* children within a zero tree */
extern unsigned char pruned[8]; /* used to keep count whether a zero tree is
pruned */

extern unsigned char total_links; /* keeps count of pruned trees */
extern unsigned char link_list[6]; /* list with unpruned trees in it */

unsigned char kids[64] = {0, 4, 8, 12, 16, 18, 24, 26, 32, 34, 40, 42, 48, 50,
56, 58, 64, 66, 68,
70, 80, 82, 84, 86, 96, 98, 100, 102, 112, 114, 116,
118, 128, 130, 132,
134, 144, 146, 148, 150, 160, 162, 164, 166, 176, 178,
180, 182, 192, 194,
196, 198, 208, 210, 212, 214, 224, 226, 228, 230, 240,
242, 244, 246 };

unsigned char max_pruned[4] = { 0, 3, 12, 48};
unsigned char childplus[4] = { 0, 2, 4, 8};
unsigned char loopind[4] = { 0, 4, 16, 64};
unsigned char level;

/*****
*
* Function : code_dlist2
* Args : none
*
* Description:
* code_dlist2 scans six zero trees in order for significant
* coefficients, it also prunes off zero trees that have no
* significant coefficients below the current level.
*
* Return Values:
* None
*
*****/

void code_dlist2()
{
int sym,i,p,s;
unsigned short t,tt,j;
unsigned char temp_links;

register unsigned int m,c;

/* Perform Dominant pass */

while (STOP == 0)
{

```

```

for (s=0;s<7;s++) pruned[s] = 0;
for (s=0;s<7;s++) pruned_children[s] = 0;

for (s=0;s<6;s++)
    comp_dbits(s);

m = 0;
for (s=0;s<6;s++)
{
    if (pruned_children[s] < 1)
    {
        link_list[m++] = s;
        pruned[s] = 1;
    }
}

total_links = m;

for (s=0;s<7;s++) pruned_children[s] = 0;

level = 1;
m = 1;

while ((m!=64) && (total_links!=0))
{
    for(;m<loopind[level];m++)
    {
        c = kids[m];
        for (i=0;i<total_links;i++)
        {
            s = link_list[i];
            if ((stats_flag[m + s*256]&2)==2)
                new_dbits_nwo(m,c,childplus[level],s);
            else if ((stats_flag[m + s*256]&6)==0)
                new_dbits_wo(m,c,childplus[level],s);
        }
    }

    temp_links = total_links;
    total_links = 0;
    for (i=0;i<temp_links;i++)
    {
        s = link_list[i];
        if (pruned_children[s] < max_pruned[level] )
        {
            link_list[temp_links++] = s;
            pruned_children[s] = 0;
        }
    }

    for (s=0;s<7;s++) pruned_children[s] = 0;

    level = level + 1;
}

if ((STOP==0) && (total_links!=0))
    for(m=64;m<256;m++)
    {
        for (i=0;i<total_links;i++)
        {

```

```

        s = link_list[i];
        if ((stats_flag[m+s*256]&6)==0)
        {
            new_dbits2(m,s);
        }
    }
}

if (sym_index>0)
    do_syms();

sym_index = 0;

start_model(2);

if (STOP==0)
    PASS_ENC += BITE;

t = THRESH>>BITE;

THRESH = THRESH>>BITE;

if (PASS_ENC == BITE)
{
    BITE = RB_ENC;
#ifdef NO_MULTI_BITPLANE
    BITE = 1;
#endif
    TMASK = THRESH;
    for(m=1;m<BITE;m++)
        TMASK = TMASK | (THRESH>>m);
}
else
{
    BITE = 1;
    TMASK = THRESH;
}

SHFT_DN -= BITE;

/* Subordinate Pass */

if ((STOP == 0) && (list_index > 0))
    subpass(t);

sym_index = 0;

start_model(1<<(BITE+1));

/* quick clear all passdown flags */

for (i=0;i<384;i++)
    stomp_flag[i] = stomp_flag[i] & 0xfdfdfdfd;

if ((THRESH & 0x03) != 0) STOP = 1;
} /* end while */
}

```

# NAWCWD TP 8442

```

/*****
**
**  diffs.s (PP Program)
**
**  Written by Jim Witham, Code 472300D, 939-3599
**
**  File that contains the following assembly language subroutines:
**
**  $diff1
**  $diff2
**  $sum1
**
*****/

.global $diff1
.global $diff2
.global $sum1

/*****
*
*  Function  : $diff1
*  Args      : none
*  Passed in :
*
*  Description:
*      $diff1 takes 16 bit coefficients and subtracts off
*      16 bit buffered coefficients, and writes the results
*      overtop of the old 16 bit coefficients.
*
*      The first set of coefficients reside in memory locations
*      $0000-$0BFE, and the buffered coefficients reside in
*      memory locations $0C00-$0FFE and $8000-$87FE.
*
*  Return Values:
*      none
*
*****/

$diff1:

    a0 = &*(dba)      ;16 bit difference data - Data Ram 0
    a1 = &*(dba) + 0xc00 ;16 bit difference data - Data Ram 1

    sr = 0xad

    le2 = loopend2
    lrs2 = 0xff

    nop
    nop

    d1 = *a0
    d2 = *a1++
    d1 =m d1 - d2
loopend2:
    *a0++ = d1

    a1 = &*(dba) + 0x8000 ;16 bit difference data - Data Ram 2

    le2 = loopend3
    lrs2 = 0x1ff

```

# NAWCWD TP 8442

```

nop
nop

d1 = *a0
d2 = *a1++
d1 =m d1 - d2
loopend3:
    *a0++ = d1

br = iprs
sr = 0x36
nop

```

```

/*****
*
*   Function   : $diff2
*   Args      : none
*   Passed in :
*
*   Description:
*       $diff2 takes 16 bit residual coefficients and does a
*       'special' subtraction from the previous buffered
*       coefficients. This special subtraction is characterized
*       by this simple pseudo-code:
*
*           if (buffer > 0) then result = buffer - residual
*           else result = buffer + residual
*           if ((buffer - residual) = 0) then result = 0
*
*       The residual resides in memory locations $0000-$0BFE,
*       and the buffer resides in memory lcoations $0C00-$0FFE
*       and $8000-$87FE.
*
*   Return Values:
*       none
*
*****/

```

\$diff2:

```

    a0 = &*(dba)      ;16 bit difference data - Data Ram 0
    a1 = &*(dba) + 0xc00 ;16 bit difference data - Data Ram 1

    le2 = loopend4a
    lrs2 = 0x1ff
    nop
    nop

    d1 =h *a0
    d2 =h *a1++
    d3 = d2 + d1
    d2 = d2 - 0
    d3 =[gt] d2 - d1
    d4 = d2 - d1
    d3 =[eq] a15
loopend4a:
    *a0++ =h d3

    a1 = &*(dba) + 0x8000 ;16 bit difference data - Data Ram 2

    le2 = loopend5a

```

# NAWCWD TP 8442

```

lrs2 = 0x3ff

nop
nop

d1 =h *a0
d2 =h *a1++
d3 = d2 + d1
d2 = d2 - 0
d3 =[gt] d2 - d1
d4 = d2 - d1
d3 =[eq] a15
loopend5a:
    *a0++ =h d3

br = iprs
sr = 0x36
nop

/*****
*
*   Function   : $sum1
*   Args      : none
*   Passed in :
*
*   Description:
*       $sum1 implements the 'leaky' integrator by taking two
*       blocks of coefficients and multiplying one by a leakage
*       factor (<1, in our case 7/8) and then summing it with
*       another block of coefficients.
*       This is characterized by this simple pseudo-code:
*
*           result = (7/8)*(input + buffer)
*
*       The residual resides in memory locations $0000-$0BFE,
*       and the buffer resides in memory locations $0C00-$0FFE
*       and $8000-$87FE.
*
*   Return Values:
*       none
*
*****/

$sum1:

a0 = &*(dba)      ;16 bit difference data - Data Ram 0
a1 = &*(dba) + 0xc00 ;16 bit difference data - Data Ram 1

le2 = loopend6
lrs2 = 0x1ff

d3 = 7
nop

d1 =h *a0
d2 =h *a1++
d1 = d1 + d2
d1 = d1 * d3
d1 = d1 >>s 3
loopend6:
    *a0++ =h d1

```



## NAWCWD TP 8442

```
a1 = &*(dba) + 0x8000    ;16 bit difference data - Data Ram 2

le2 = loopend7
lrs2 = 0x3ff

nop
nop

d1 =h *a0
d2 =h *a1++
d1 = d1 + d2
d1 = d1 * d3
d1 = d1 >>s 3
loopend7:
    *a0++ =h d1

br = iprs
sr = 0x36
nop
```

```

/*****
**
** hvenc.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $subdec_vert
** $subdec_horiz
**
*****/

.global $subdec_vert
.global $subdec_horiz

.align 2048
/*****
*
* Function : $subdec_vert
* Args : outerloop,innerloop
* Passed in : (d1 ,d2)
*
* Description:
*   outerloop - the width of the coefficient patch
*
*   innerloop - the height of the coefficient patch
*
*   $subdec_vert will be called to perform the wavelet
*   decomposition in the vertical direction.
*
* Return Values:
*   None
*
*****/

$subdec_vert:
    lctl = 0x0 ;reset looping capability

    lr0 = d1 - 1 ;outerloop
    lr1 = d2 - 3 ;innerloop
    a4 = d1

    d7 = 0 ; k = 0

; Set up zero-overhead loops
    le1 = InnerLoopEnd ;
    ls1 = InnerLoop ;

    le0 = OuterLoopEnd
    ls0 = OuterLoop
    nop
    lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

    d3 = a4
    d4 = d3 + d3
    x1 = d4
    d4 = d4 + d3
    x2 = d4

```

# NAWCWD TP 8442

OuterLoop:

```
;>>>>          img[index][k]      =      (img[index][k]>>1)      -
((img[0][k]+img[2*index][k])>>2);

      nop
      x0 = d7
      nop
      a0 =h *(dba + [x0])
      nop
      x0 = a4

      d2 =sh *(a0 + [x1])
      d3 =sh *(a0)

      d4 =sh *(a0 + [x0])
||      d2 = d2 + d3

      d4 = (d4 >>s 1)
      d4 = d4 - (d2 >>s 2)
      *(a0 + [x0]) =h d4

;>>>>          img[0][k] += img[index][k];

      d3 = d3 + d4
      *(a0++=[x0]) =h d3
      nop
```

InnerLoop:

```
;img[2*1+index][k]      =      (img[2*1+index][k]>>1)      -
((img[2*1][k]+img[2*1+2*index][k])>>2);

      d3 =sh *(a0 + [x0])
      d4 =sh *(a0 + [x2])

      d2 =sh *(a0 + [x1])
||      d5 = d3 + d4

      d2 = (d2 >>s 1)
||      d4 =sh *(a0++=[x0])
      d2 = d2 - (d5 >>s 2)

      *(a0 + [x0]) =h d2
||      d5 = d2 + d4

;img[2*1][k] += ((img[2*1+index][k]+img[2*1-index][k]+2)>>1);

      d3 = d3 + (d5 >>s 1)
```

InnerLoopEnd:

```
      *(a0++=[x0]) =h d3

;>>>>          img[YSIZE-index][k]      =      (img[YSIZE-index][k]-img[YSIZE-
2*index][k])>>1;

      d2 =sh *(a0 + [x1])
      d3 =sh *(a0 + [x0])
      d2 = d2 - d3
      d2 = (d2 >>s 1)
      *(a0 + [x1]) =h d2
```

# NAWCWD TP 8442

```

; >>>>      img[YSIZE-2*index][k]  += ((img[YSIZE-index][k]+img[YSIZE-
3*index][k]+2)>>1);

      d4 =sh *(a0)
;      || d2 = d2 + 2
      d5 = d2 + d4
      d3 = d3 + (d5 >>s 1)

      *(a0 + [x0]) =h d3

OuterLoopEnd:
      d7 = d7 + 1

      br = iprs
      nop
      nop

/*****
*
*   Function   : $subdec_horiz
*   Args      : outerloop,innerloop
*   Passed in : (d1      ,d2)
*
*   Description:
*       outerloop - the width of the coefficient patch
*
*       innerloop - the height of the coefficient patch
*
*       $subdec_horiz will be called to perform the wavelet
*       decomposition in the horizontal direction.
*
*   Return Values:
*       None
*
*****/

$subdec_horiz:

      lctl = 0x0 ;reset looping capability

;set loop reload, counter
      lr0 = d1 - 1
      lr1 = d2 - 3

      a4 = d2          ; innerloop

      d7 = 0          ; k = 0

;   Set up loop to iterate ((128 >> scale) - 2) times
      le1 = InnerLoopEnd2      ;
      ls1 = InnerLoop2         ;

      le0 = OuterLoopEnd2     ;
      ls0 = OuterLoop2        ;

      nop

      lctl = 0xa9 ;associate le0 with lc0 and le1 with lc1

      nop
      nop

```

# NAWCWD TP 8442

OuterLoop2:

```
;>>>>      img[k][index] -= ((img[k][0]+img[k][2*index])>>1);
;              a0              a1              a2
```

```
    d3 = a4
    d3 = d3 * d7
    d3 = d3 << 1
    x0 = d3
    nop
    a0 =h &*(dba + [x0])
    nop
```

```
    d3 =sh *(a0)
    d2 =sh *(a0 + [2])
```

```
||      d4 =sh *(a0 + [1])
        d2 = d2 + d3
```

```
    d4 = d4 - (d2 >>s 1)
    *(a0 + [1]) =h d4
```

```
;>>>>      img[k][0] = (img[k][0]<<1) + img[k][index];
```

```
    d4 = d4 + (d3 << 1)
    *(a0++=[1]) =h d4
```

```
    nop
```

InnerLoop2:

```
;img[k][2*1+index] -= ((img[k][2*1]+img[k][2*1+2*index])>>1);
```

```
    d3 =sh *(a0 + [1])
    d4 =sh *(a0 + [3])
```

```
    d2 =sh *(a0 + [2])
|| d5 = d3 + d4
```

```
    d2 = d2 - (d5 >>s 1)
|| d4 =sh *(a0++=[1])
```

```
    *(a0 + [1]) =h d2
|| d5 = d2 + d4
```

```
;img[k][2*1] = (img[k][2*1]<<1) + ((img[k][2*1+index]+img[k][2*1-index])>>1);
```

```
    d3 = d3 << 1
    d3 = d3 + (d5 >>s 1)
```

InnerLoopEnd2:

```
    *(a0++=[1]) =h d3
```

```
;>>>>      img[k][XSIZE-index] -= img[k][XSIZE-2*index];
```

```
    d2 =sh *(a0 + [2])
    d3 =sh *(a0 + [1])
    d2 = d2 - d3
    *(a0 + [2]) =h d2
```

```
;>>>>      img[k][XSIZE-2*index] = (img[k][XSIZE-2*index]<<1) +
    ((img[k][XSIZE-index]+img[k][XSIZE-3*index])>>1);
```

# NAWCWD TP 8442

```
        d4 =sh *(a0)
;      || d2 = d2 + 2
        d5 = d2 + d4
        d3 = d3 << 1
        d3 = d3 + (d5 >>s 1)

        *(a0 + [1]) =h d3

OuterLoopEnd2:
        d7 = d7 + 1

        br = iprs
        nop
        nop
;      branch occurs here
```

# NAWCWD TP 8442

```

/*****
**
** j_enc.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** PP Program that orchestrates the generation of the bitstream.  Processes
** 6 Zero Trees per partition.
**
*****/

#include <mvp.h>
#include "arith.h"
#include "modlp.h"

#define ASSEM
#define ASSEM3
#define ASSEM2

/* #define NO_MULTI_BITPLANE */

#define XSIZE 512
#define YSIZE 256
#define NSCALES 5
#define AX 16 /* = XSIZE/BS */
#define AY 8 /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

#define maxsyms 240

/* REAL-TIME: PROCESSES 2 ZEROTREES PER PARTITION: TRUNCATED */

unsigned char emubrk;

extern short *img;
extern short *coeff_block;
extern short *stats_val;
extern short *ztl;
extern unsigned char *stats_flag;
extern unsigned int *stomp_flag;
extern unsigned int *stomp_ztl;
extern unsigned char *byte_stream;
extern unsigned char *tbuf;

extern unsigned short *list;
extern unsigned short list_index;

extern short *qstats_val;
extern unsigned short *qindex;

extern unsigned char qcount;

extern int No_of_chars;

extern int EOF_symbol; /* Index of EOF symbol */

extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

```

```

extern short int cum_freq[Max_No_of_symbols+1];

extern int buffer_index; /* JCW */
extern unsigned short bit_index;

extern unsigned short T_BYTES;

extern short maxval;

typedef struct pstr
{
    unsigned char d,i;
} PSTR;

/* Each element can be read as needed */

extern unsigned short *ALLOC;

extern int FIRST_ENC,FCNT_ENC; /* 1st pass bite size */

extern unsigned char *passes_d;

/* Required on-line storage for high speed */

extern int STOP;
extern int BYTE_CNT;
extern unsigned short THRESH;
extern int BITE;
extern int SIG_COEF;
/* extern int SIG[15]; */

unsigned char PASS_ENC;
unsigned short TMASK;

    int SHFT_DN,RB_ENC;
    short BYTE_TOTAL;

extern int whoami();

unsigned char buffer; /* Bits buffered for output */
unsigned char bits_to_go; /* # bits free in buffer */

unsigned char pruned_children[8]; /* used to keep count of the number of pruned
*/
/* children within a zero tree */
unsigned char pruned[8]; /* used to keep count whether a zero tree is pruned */
unsigned char total_links; /* keeps count of pruned trees */
unsigned char link_list[6]; /* list with unpruned trees in it */

void writeout(int sym);

void start_model(int nchars);
void update_model(int symbol);

void encode_symbol(int symbol);
void bit_plus_follow(int bit);
void output_bit(int bit);

void new_dbits(int m, int c, int number, int s);
void new_dbits2(int m, int s);

```



# NAWCWD TP 8442

```

void comp_ztr(int s);

void subpass(int t);

void diff1();
void diff2();
void sum1();
void sum2();

/* CURRENT STATE OF ENCODING */

int low, high; /* Ends of current code region */
short bits_to_follow; /* Number of opposite bits to output
                        after the next bits */

extern shared int local_maxval[4];
extern shared int global_maxval;

/* int time_cdwo,time_cdnwo,time_cd2wo,time_sym; */
/* int time_cd2nwo,time_spwo,time_spnwo; */

unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

/* START ENCODING A STREAM OF SYMBOLS */

void start_encoding()
{
    low = 0; /* Full code range */
    high = Top_value;
    bits_to_follow = 0; /* No bits to follow */
}

void writeout(sym)
int sym;
{
    if (BYTE_CNT++ < T_BYTES)
    {
        byte_stream[buffer_index++] = sym; /* Output filled buffer JCW*/
        BYTE_TOTAL++;
    }
    else
        STOP = 1;
}

/* INITIALIZE FOR BIT OUTPUT */

void start_outputting_bits()
{
    buffer = 0; /* Buffer is empty at start */
    bits_to_go = 8;
}

/* *****
                        INPUT_BLOCK
***** */
void input_block(s,p)
int s,p;
{

```

```

int i,j,k,l;

i = s * 256;
for(k=NSCALES-2;k>0;k--)
{
    if (k==(NSCALES-2))
        for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
            for(l=0;l<(1<<NSCALES);l+=(2<<k))
                stats_val[i++] = coeff_block[p*512 + j*32+l];

    for(j=0;j<(1<<(NSCALES-1));j+=(1<<k))
        for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
            stats_val[i++] = coeff_block[p*512 + j*32+l];

    for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
        for(l=0;l<(1<<NSCALES);l+=(2<<k))
            stats_val[i++] = coeff_block[p*512 + j*32+l];

    for(j=(1<<(k-1));j<(1<<(NSCALES-1));j+=(1<<k))
        for(l=(1<<k);l<(1<<NSCALES);l+=(2<<k))
            stats_val[i++] = coeff_block[p*512 + j*32+l];

}

}

unsigned char sym_array[256];
unsigned short sym_index;

/* *****
                        COMP_DBITS
***** */
void comp_dbits(s)
int s;

{
    int sym,tflg,apx,t,m0;
    int i,m;

    m = s * 256;
    if ((stats_flag[s * 256]&6)==0)
    {
        tflg = ((TMASK&zt1[s * 64]) == 0);
        t = (TMASK&abs(stats_val[m]))>>SHFT_DN;
        sym = t + (((t!=0)&&(stats_val[m]>0)) ? ((1<<BITE)-1):0) + ((~tflg)&1);

        if ((t>0)&&(list_index<254))
            list[list_index++] = m;

        sym_array[sym_index++] = sym;

        if (t>0)
        {
            qstats_val[qcount] = abs(stats_val[m]) - (((t*THRESH)>>(BITE-1))+(THRESH>>BITE));
            qindex[qcount++] = m;
        }

        if (tflg!=0)
        {
            pruned_children[s] = 1;
            stats_flag[m + 1] = stats_flag[m + 1] | (tflg<<1);
        }
    }
}

```

# NAWCWD TP 8442

```

        stats_flag[m + 2] = stats_flag[m + 2] | (tflg<<1);
        stats_flag[m + 3] = stats_flag[m + 3] | (tflg<<1);
    }

    stats_flag[m] = (t!=0)<<2;
}
else if ((stats_flag[m]&2)!=0)
{
    stats_flag[m] = stats_flag[m] & 252;
    stats_flag[m + 1] = stats_flag[m + 1] | 2;
    stats_flag[m + 2] = stats_flag[m + 2] | 2;
    stats_flag[m + 3] = stats_flag[m + 3] | 2;
    pruned_children[s] = 1;
}
}

/* ***** */

void comp_dbits2(m)
    int m;

{
    int sym,t,tflg;

    t = (TMASK&abs(stats_val[m]))>>SHFT_DN;
    sym = t + (((t!=0)&&(stats_val[m]>0)) ? ((1<<BITE)-1):0) + (t!=0);

    /* writeout(sym); */

    sym_array[sym_index++] = sym;

    if (sym_index > maxsyms)
    {
        do_syms();
        sym_index = 0;
    }

    /*      stats_val[m] = (t>0) ? (abs(stats_val[m]) - ((t*THRESH)/(1<<(BITE-
1))+THRESH/(2<<(BITE-1)))) : stats_val[m]; */
    stats_val[m] = (t>0) ? (abs(stats_val[m]) - ((t*THRESH)>>(BITE-
1)) + (THRESH>>BITE))) : stats_val[m];
    stats_flag[m] = (t!=0)<<2;
}

```

# NAWCWD TP 8442

```

/* *****
CODE_DLIST
***** */

void code_dlist()
{
    int sym,i,p,s;
    unsigned short t,tt,j;
    short saveme;
    unsigned char savemec;

    register unsigned int m,c;

    /* Perform Dominant pass */

    for (s=0;s<6;s++)
        comp_dbits(0,1,2,3,s);

    if (STOP==0)
        for(m=1;m<4;m++)
        {
            c = 4*m;
            for (s=0;s<6;s++)
            {
                #ifdef ASSEM
                    if ((stats_flag[m + s*256]&2)==2)
                        new_dbits_nwo(m,c,2,s);
                    else if ((stats_flag[m + s*256]&6)==0)
                        new_dbits_wo(m,c,2,s);
                #else
                    if ((stats_flag[m + s*256]&6)!=4)
                        comp_dbits(m,c,c+1,c+2,c+3,s);
                #endif
            }
        }

    if (STOP==0)
        for(m=4;m<16;m++)
        {
            c = 8*(m/2) + 2*(m%2);
            for (s=0;s<6;s++)
            {
                #ifdef ASSEM
                    if ((stats_flag[m + s*256]&2)==2)
                        new_dbits_nwo(m,c,4,s);
                    else if ((stats_flag[m + s*256]&6)==0)
                        new_dbits_wo(m,c,4,s);
                #else
                    if ((stats_flag[m + s*256]&6)!=4)
                        comp_dbits(m,c,c+1,c+4,c+5,s);
                #endif
            }
        }

    if (STOP==0)
        for(m=16;m<64;m++)
        {
            c = 16*(m/4) + 2*(m%4);
            for (s=0;s<6;s++)
            {

```

```

#ifdef ASSEM
    if ((stats_flag[m + s*256]&2)==2)
        new_dbits_nwo(m,c,8,s);
    else if ((stats_flag[m + s*256]&6)==0)
        new_dbits_wo(m,c,8,s);
#else
    if ((stats_flag[m + s*256]&6)!=4)
        comp_dbits(m,c,c+1,c+8,c+9,s);
#endif
}
}

/* Eliminate if NSCALES = 3 */

if (STOP==0)
    for(m=64;m<256;m++)
    {
        for (s=0;s<6;s++)
        {
            if ((stats_flag[m+s*256]&2)==2)
            {
                stats_flag[m+s*256] = stats_flag[m+s*256] & 252;
            }
            else if ((stats_flag[m+s*256]&6)==0)
            {
#ifdef ASSEM3
                new_dbits2(m,s);
            #else
                comp_dbits2(m+s*256);
            #endif
            }
        }
    }

if (sym_index>0)
    do_syms();

sym_index = 0;

start_model(2);

if (STOP==0)
    PASS_ENC += BITE;

t = THRESH>>BITE;

THRESH = THRESH>>BITE;

if (PASS_ENC == BITE)
{
    BITE = RB_ENC;
#ifdef NO_MULTI_BITPLANE
    BITE = 1;
#endif
    TMASK = THRESH;
    for(m=1;m<BITE;m++)
        TMASK = TMASK | (THRESH>>m);
}
else

```

```

    {
        BITE = 1;
        TMASK = THRESH;
    }

    SHFT_DN -= BITE;

/* Subordinate Pass */

    subpass(t);

    if (sym_index>0)
        do_syms();

    sym_index = 0;

    start_model(1<<(BITE+1));
}

/* *****
   P_CODE
   ***** */

void p_code()

{
    int k,l,i,j,mask,bite,pmin; /* maxval */

    int aa,ab;

/* DEFINE THIS */

    BYTE_TOTAL = 12;

    sym_index = 0;

aa = 0xffffffff; ab = 0x00;

/* Compute mask */

    FIRST_ENC = tbuf[0];

    if (FIRST_ENC == 6)
    {
        bite = 3;
        RB_ENC = 3;
    }
    else if (FIRST_ENC == 5)
    {
        bite = 3;
        RB_ENC = 2;
    }
    else if (FIRST_ENC==4)
    {
        bite = 2;
        RB_ENC = 2;
    }
    else
    {

```

```

    bite = FIRST_ENC;
    RB_ENC = 1;
}
#ifdef NO_MULTI_BITPLANE
BITE = 1;
#endif

pmin = 100;

maxval = 0;

for (j=0;j<10;j++)
{
    for (i=0;i<6;i=i+2)
    {
/* wait for new coefficients */
        while((INTFLG&(1<<20))==0);
        INTFLG = 1<<20;

        input_block(i,0);
        input_block(i + 1,1);

/* tell MP done inputting this block of coefficients */
        asm("    x2 = 0x00002100");
        asm("    cmnd = x2");
    }

/* wait for buffer to be written in */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

/* memory = coefficients - buffer */
    diff1();

/* determine maxval */

    for (k=0;k<1536;k++)
    {
        if (abs(stats_val[k]) > maxval)
            maxval = abs(stats_val[k]);
    }

    local_maxval[whoami()] = maxval;

/* tell MP done with differencing and maxval calculation */

    asm("    x2 = 0x00002100");
    asm("    cmnd = x2");

} /* end for j */

/* wait for global maxval computation to complete */

while((INTFLG&(1<<20))==0);
INTFLG = 1<<20;

maxval = 1<<global_maxval;

mask = maxval;
for(k=1;k<bite;k++)
    mask = mask | (maxval>>k);

```

# NAWCWD TP 8442

```

/* Main Loop: Continue until stop condition is reached */
while (*pp_stop_encode == 0)
{
/* emubrk = 1; */

    BITE = bite;
#ifdef NO_MULTI_BITPLANE
    BITE = 1;
#endif
    PASS_ENC = 0;
    STOP = 0;
    BYTE_CNT = 0;

    THRESH = maxval;
    SHFT_DN = global_maxval - BITE + 1;
    TMASK = mask;
    buffer_index = 0;

    list_index = 0;

    start_model(1<<(BITE+1));
    start_outputing_bits();
    start_encoding();

    /* wait for new coefficients */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    if (*pp_stop_encode == 0)
    {
        for (i=0;i<6;i++)
        {
            comp_ztr(i);
        }

        T_BYTES = ALLOC[0];
        for (bit_index=0;bit_index<T_BYTES;bit_index++) byte_stream[bit_index] =
0;
        bit_index = 0;

        for(j=0;j<384;j++)
            stomp_flag[j] = 0;

        emubrk = 0x21;

        code_dlist2();

        emubrk = 0x22;

/* emubrk = 3; */

        if (ALLOC[0] <3)
            passes_d[0] = 0;
        else if (ALLOC[0]==3)
            passes_d[0] = PASS_ENC-4;
        else if (ALLOC[0]<7)
            passes_d[0] = PASS_ENC-2;
        else
        {

```



# NAWCWD TP 8442

```

    passes_d[0] = PASS_ENC-1;
    if (PASS_ENC-1 < pmin)
        pmin = PASS_ENC-1;
    }

    /* tell MP done with this block of pixels */

    asm("    x2 = 0x00002100");
    asm("    cmnd = x2");

    /* wait for coefficients to be written in */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    /* special differencing */
    /* memory = coefficients before quant. - leftover coeff after quant */
    /* 0x0000 = 0x0c00->0x0ffe          - 0x0000->0xbfe          */
    diff2();

    /* tell MP done with the differencing */

    asm("    x2 = 0x00002100");
    asm("    cmnd = x2");

    /* wait for coefficients to be written in */
    while((INTFLG&(1<<20))==0);
    INTFLG = 1<<20;

    sum1();

    /* tell MP done with the summing */

    asm("    x2 = 0x00002100");
    asm("    cmnd = x2");

    } /*end if */
} /* end while */

/* Transmit next BITE size */

if (pmin==0)
    FIRST_ENC -= 1;
else if ((pmin > FIRST_ENC)&&(FCNT_ENC > 2)&&(pmin<7))
{
    FCNT_ENC = 0;
    FIRST_ENC = pmin-1;
}
else if (pmin > FIRST_ENC)
    FCNT_ENC++;
else if (pmin<=FIRST_ENC)
{
    FIRST_ENC = pmin;
    FCNT_ENC = 0;
}

tbuf[1] = FIRST_ENC;
}

```

# NAWCWD TP 8442

```

/*****
**
** main.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Main PP Program that calls all the other routines to perform the wavelet
** decomposition and bitstream generation.
**
*****/

#include "modlp.h"

#define XCOLS 8      /* Max Columns that can be held in internal memory */
#define YROWS 4      /* Max Columns that can be held in internal memory */

#define XSIZE 512    /* Max Image Size */
#define YSIZE 240    /* Max Image Height */
#define ML 50        /* Max order of filter allowed */
#define NSCALES 4
#define AX 16 /* = XSIZE/BS */
#define AY 8  /* = YSIZE/BS */
#define BS 32 /* = 2^NSCALES */

#define ENCODE
#define ENCODE_STREAM

#define DISPLAY

    unsigned short T_BYTES;
    unsigned char *pic;
    short *img;

    short *stats_val;
    short *stats_apx;
    unsigned char *stats_flag;
    unsigned int *stomp_flag;
    unsigned int *stomp_ztl;
    unsigned char *byte_stream;
    short *coeff_block;
    unsigned short *ztl;
    unsigned char *tbuf;
    unsigned short *ALLOC;

    unsigned short *list;

    short *qstats_val;
    unsigned short *qindex;

unsigned char qcount = 0;

int No_of_chars;

int EOF_symbol; /* Index of EOF symbol */

int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

unsigned char char_to_index[Max_No_of_chars]; /* JCW old version was int */
unsigned char index_to_char[Max_No_of_symbols+1]; /* JCW old version was int */

```

# NAWCWD TP 8442

```

short int cum_freq[Max_No_of_symbols+1]; /* JCW old version was int */

int buffer_index; /* JCW */

extern void subdec_vert(int outerloop,int innerloop);
extern void subdec_horiz(int outerloop,int innerloop);

extern int calc_n_sub_mean(int oldmean);
extern void add_n_conv8(int oldmean);

/* extern void subsyn_vert(int outerloop,int innerloop);
extern void subsyn_horiz(int outerloop,int innerloop); */

extern int whoami();

/* ***** MAIN PROGRAM ***** */

register extern volatile unsigned int INTFLG;

short column_outerloop[5] = { 8, 16, 32, 16, 8 };
short row_outerloop[5]    = { 4, 8, 16, 8, 4 };

short column_innerloop[5] = { 120, 60, 30, 15, 8 };
/* short column_innerloop[5] = { 59, 29, 14, 6, 8 }; */
short row_innerloop[5]    = { 256, 128, 64, 32, 16 };

unsigned char vert_loop_index[6] = { 16, 4, 1, 1, 1 };
unsigned char horiz_loop_index[6] = { 15, 5, 1, 1, 1 };

short maxval;

extern shared int mean_pp[4];
extern shared int global_mean;

unsigned char *passes_d;

/* extern int RB_ENC,RB_DEC; */
extern unsigned char PASS_ENC,PASS_DEC;

int STOP;
int BYTE_CNT;
unsigned short THRESH;
int BITE;
int SIG_COEF;
/* int SIG[15]; */

int FIRST_ENC,FIRST_DEC;
int FCNT_ENC,FCNT_DEC;

unsigned int getaway_address;
unsigned char quick_getaway = 0;

main()
{
    int k,l;
    int mean;

```

```
/* initialize pic[], img[], stats_val[], stats_flag[], byte_stream[], and
coeff_block[] pointers */
```

```
asm("    d1 = &(dba)");
asm("    *(xba+$pic) = d1");
asm("    *(xba+$img) = d1");
asm("    *(xba+$stats_val) = d1");
asm("    *(xba+$stats_apx) = d1");
asm("    d1 = &(dba + 0x8000)");
asm("    *(xba+$stats_flag) = d1");
asm("    *(xba+$stomp_flag) = d1");
asm("    *(xba+$coeff_block) = d1");
asm("    x0 = 0x8602");
asm("    nop");
asm("    d1 = &(dba + x0)");
asm("    *(xba+$list) = d1");
asm("    d1 = &(dba + 0xc00)");
asm("    *(xba+$ztl) = d1");
asm("    *(xba+$stomp_ztl) = d1");
asm("    d1 = &(pba + 0x630)");
asm("    *(xba+$byte_stream) = d1");
asm("    d1 = &(pba + 0x620)");
asm("    *(xba+$passes_d) = d1");
asm("    d1 = &(pba + 0x5fc)");
asm("    *(xba+$tbuf) = d1");
asm("    d1 = &(pba + 0x600)");
asm("    *(xba+$ALLOC) = d1");
asm("    d1 = &(pba + 0x100)");
asm("    *(xba+$qstats_val) = d1");
asm("    d1 = &(pba + 0x180)");
asm("    *(xba+$qindex) = d1");
```

```
FIRST_ENC = 4;
FIRST_DEC = 4;
```

```
tbuf[0] = FIRST_DEC;
```

```
FCNT_ENC = 0;
FCNT_DEC = 0;
```

```
/* Clear the message interrupt flag that comes from the MP, just in case */
```

```
INTFLG = 1<<20;
quick_getaway = 0;
```

```
while (1)
{
```

```
#ifdef ENCODE
```

```
    mean = 0;
```

```
/* Decompose image */
```

```
for(k=0;k<(NSCALES+1);k++)
{
    for(l=0;l<horiz_loop_index[k];l++)
    {
        /* wait for new pixels */

        while((INTFLG&(1<<20))==0);
```

# NAWCWD TP 8442

```

    INTFLG = 1<<20;

    if (k==0)
        mean = mean + calc_n_sub_mean(global_mean);

    subdec_horiz(row_outerloop[k],row_innerloop[k]);

    /* tell MP done with this block of pixels */

    asm("    d7 = 0x00002100");
    asm("    cmnd = d7");

    }

/* Put out local mean for mp to do global mean calculation */

    mean_pp[whoami()] = mean;

    if (k!=NSCALES)
        for(l=0;l<vert_loop_index[k];l++)
        {
            /* wait for new pixels */

            while((INTFLG&(1<<20))==0);

            INTFLG = 1<<20;

            subdec_vert(column_outerloop[k],column_innerloop[k]);

            /* tell MP done with this block of pixels */

            asm("    d7 = 0x00002100");
            asm("    cmnd = d7");

        }

    }

#endif

/* Fake it for now */
/* maxval = 0x800<<3; */

/* Form data stream */

#ifdef ENCODE_STREAM
    p_code();
#endif

}

}

```

# NAWCWD TP 8442

```

/*****
**
** mean2.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $calc_n_sub_mean
** $whoami
**
*****/

```

```

.global $calc_n_sub_mean
.global $whoami

```

```

/*****
*
* Function : $calc_n_sub_mean
* Args : oldmean
* Passed in : d1
*
* Description:
* oldmean - the mean on the previous image
*
* $calc_n_sub_mean will be called to both calculate the sum
* of the pixels in this patch, and to subtract off the
* previous images mean (performing an 8 bit minus and 16
* bit subtraction, yielding a 16 bit result, which is then
* further processed by shifting it to the left 2 places.
*
* Return Values:
* d5 - returns the sum of this patch of pixels.
*
*****/

```

\$calc\_n\_sub\_mean:

```

    d5 = 0
    a8 = &(dba) + 4094
    a0 = &(dba) + 2047

    d1 = 0 - (d1 << 2)          ; make mean so it can be subtracted

    le2 = loopend
    lrs2 = 510

    d2 =ub *a0--
    d5 = d5 + d2

;    loop starts

    x1 = d1 + (d2 << 2)
|| d2 =ub *a0--
   *a8-- =h x1
|| d5 = d5 + d2

    x1 = d1 + (d2 << 2)
|| d2 =ub *a0--
   *a8-- =h x1
|| d5 = d5 + d2

```

# NAWCWD TP 8442

```

    x1 = d1 + (d2 << 2)
    || d2 =ub  *a0--
    *a8-- =h  x1
    || d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    || d2 =ub  *a0--
loopend:
    *a8-- =h  x1
    || d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    || d2 =ub  *a0--
    *a8-- =h  x1
    || d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    || d2 =ub  *a0--
    *a8-- =h  x1
    || d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    || d2 =ub  *a0--
    *a8-- =h  x1
    || d5 = d5 + d2

    x1 = d1 + (d2 << 2)
    *a8-- =h  x1

```

```

br = iprs
nop
nop

```

```

/*****
*
*   Function   : $whoami
*   Args      : none
*   Passed in  :
*
*   Description:
*       $whoami will be called to determine which PP this is.
*
*   Return Values:
*       d5 - returns the PP number assigned to this PP.
*
*****/

```

\$whoami:

```

    d5 = comm & 0x03
br = iprs
nop
nop

```

# NAWCWD TP 8442

```

/*****
**
** modlp.c (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** Contains start_model subroutine. This subroutine initializes
** the translation tables and the frequency counters for the
** arithmetic coder.
**
*****/

/* ADAPTIVE SOURCE MODEL */

#include "modlp.h"
#include "arith.h"

short int freq[Max_No_of_symbols+1]; /* Symbol frequencies */

extern int No_of_chars;

extern int EOF_symbol; /* Index of EOF symbol */

extern int No_of_symbols; /* total # of symbols */

/* TRANSLATION TABLES BETWEEN CHARS AND SYMBOL INDEXES */

extern unsigned char char_to_index[Max_No_of_chars];
extern unsigned char index_to_char[Max_No_of_symbols+1];

extern short int cum_freq[Max_No_of_symbols+1];

extern int low,high;
extern short bits_to_follow;
extern unsigned char bits_to_go;
extern unsigned char buffer;
extern int BYTE_CNT;
extern unsigned short T_BYTES;
extern unsigned char *byte_stream;
extern int buffer_index;
extern short BYTE_TOTAL;
extern int STOP;

/* OUTPUT A BIT */

void output_bit(bit)
int bit;
{
    buffer >>= 1; /* Put bit in top of buffer */
    if (bit)
        buffer |= 0x80;
    bits_to_go -= 1;
    if (bits_to_go == 0)
    {
        if (BYTE_CNT++ < T_BYTES)
        {
            byte_stream[buffer_index++] = buffer; /* Output filled buffer JCW*/
            BYTE_TOTAL++;
        }
        else
        {
            STOP = 1;
            bits_to_go = 8;
        }
    }
}

```



```

    }
}

unsigned short bit_index;

/*  INITIALIZE THE MODEL  */

void start_model(nchars)
    int nchars;
{
    int i;

/*  Initialize number of chars  */

    No_of_chars = nchars;
    No_of_symbols = nchars + 1;

/*  Setup translation tables  */

    for(i=0;i<No_of_chars;i++)
    {
        char_to_index[i] = i+1;
        index_to_char[i+1] = i;
    }

/*  Initialize frequency counts  */

    for(i=0;i<=No_of_symbols;i++)
    {
        freq[i] = 1;
        cum_freq[i] = No_of_symbols-i;
    }
    freq[0] = 0;
}

```

## NAWCWD TP 8442

```
/* INTERFACE TO THE MODEL */  
  
/* Set of symbols that may be encoded */  
  
#define Max_No_of_chars 16  
#define Max_No_of_symbols 17  
  
/* Cumulative Frequency Table */  
  
#define Max_frequency 75 /* 16383      Maximum allowed frequency cnt  $2^{14}-1$  */
```

# NAWCWD TP 8442

```

/*****
**
**
** mp.c (MP Program)
**
** Written by Jim Witham, Code 472330D, 927-1440
**
** MP Program that orchestrates data movement and kicks off PP's to implement
** the balanced wavelet algorithm written by Chuck Creusere.
**
** Interframe Encode version
*****/

*/

#include <stdlib.h>
#include <stdio.h>
#include <mvp.h>

#include <mvp_hw.h>
#include <mp_ptreq.h>

#include "icl.h"
#include "vil24.h"
#include "vol.h"
#include "bgl.h"
#include "pcic80.h"
#include "cil.h"

/* define compression ratio. Ratio is actually #:1 compression */

#define HOST

/* #define variable_compression */

#define compression 40

#define CAMERA
/* #define INTERLEAVE_CAMERA */
#define CHANNEL_LIMIT

#define ENCODE
#define ENCODE_STREAM

#define DISPLAY
#define SHOWDISPLAY

/* #define INTER_DISPLAY */

/* #define OLD_DISPLAY */

#define headersize 0

/* #define DEBUG */

#define XSIZE 512
#define YSIZE 240

#define AX 16
#define AY 15

```

```

/*
** Define a bit mask for host request bit
*/
#define SigBit(x)          (((UINT32)1)<<(x))
#define HostRequestBitMask SigBit(8)
#define FrameDoneBit 8

/*****
extern int *number_pixels;

#pragma DATA_SECTION(Semaphore,"mp_vars")
long Semaphore;

#pragma DATA_SECTION(encode_time,"mp_vars")
#pragma DATA_SECTION(decode_time,"mp_vars")
int encode_time,decode_time,proc_time;

#pragma DATA_SECTION(start_time,"mp_vars")
unsigned int start_time;

#pragma DATA_SECTION(mean_pp,"sh_vars")
#pragma DATA_SECTION(global_mean,"sh_vars")

    shared int mean_pp[4];
    shared int global_mean;

#pragma DATA_SECTION(local_maxval,"sh_vars")
#pragma DATA_SECTION(global_maxval,"sh_vars")

    shared int local_maxval[4];
    shared int global_maxval;

typedef struct pstr
{
    unsigned char d,i;
} PSTR;

#pragma DATA_SECTION(local_alloc,"mp_vars")
unsigned short local_alloc[AX*AY/6];

#pragma DATA_SECTION(comp_table,"mp_vars")
unsigned char comp_table[5] = {10, 20, 40, 80, 100};

#pragma DATA_SECTION(last_comp,"mp_vars")
unsigned char last_comp = 2;

#pragma DATA_SECTION(compression_ratio,"mp_vars")
/* UINT32 *compression_ratio = (UINT32 *) 0x010107D0; */
UINT32 *compression_ratio;

static void SignalHandler(UINT32 Signals);
static void init_alloc_table(UINT32 index);
static void init_alloc_table2(UINT32 index);

unsigned char pptime[128];
unsigned char pptask[128];
int pptime[128];
int ppstarttime = 0;
unsigned char ppindex = 0;

```

# NAWCWD TP 8442

```

/*****/

void task (void *arg)
{
    unsigned char times=0;
    unsigned char flip = 0;
    unsigned int stream_addr[2];
    PCIC80STAT ReturnVal;

    PVIL24 pVil24;
    ICL_IMG *vim,*f0,*f1;
    unsigned int dx,dy;
    int i,j;

    int lp;
    unsigned char vert_loop_index[6],horiz_loop_index[6];
    int jump_col[6];
    long jump_row[6];

    long *unused_pixels;

    long *ptr;          /* temp pointer          */
    PTREQ *p[10];       /* temp pointer to packet transfer structure */

    int temp_maxval;

/* JCWtest */

    char string1[32];
    int temp_str;
    float temp_time;

    int junki;
    unsigned short junkfill = 0;
    unsigned short * fillme = (unsigned short *) 0x90320000;

    int k,l;
    long templ;
    unsigned char tbuf;

    int t_alloc;
    int p_alloc;
    int r_alloc;

    unsigned char * tbuf_pp0      = (unsigned char *) 0x010005fc;
    unsigned char * tbuf_pp1      = (unsigned char *) 0x010015fc;
    unsigned char * tbuf_pp2      = (unsigned char *) 0x010025fc;
    unsigned char * tbuf_pp3      = (unsigned char *) 0x010035fc;

    unsigned char min_first;

    unsigned char ppnexttask[4];
    unsigned short ppallocc[4];
    unsigned char current_block;
    unsigned int table_pointer = 0x80380000;
    unsigned int table_address[4];
    unsigned short table_size[4];
    unsigned char pprequesting;
    unsigned char ppinfo[4];
    unsigned char ppdone;

```

# NAWCWD TP 8442

```

unsigned char *pp_stop_encode = (unsigned char *) 0x010007D4;

unsigned char DONE1;

float channel_bandwidth;
float bits_per_frame;
float fps;
float secs_between_frames;
float ticks_between_frames;
unsigned int time_wait;

#ifdef INTERLEAVE_CAMERA
    unsigned int started_acq;
    unsigned int done_acq;
    unsigned int line_acq;
    unsigned int last_status;
    unsigned int this_status;
#endif

UINT32 temp_ulong;
UINT32 save_p7_src;

Semaphore = 0;
comp_table[0] = 10;
comp_table[1] = 20;
comp_table[2] = 40;
comp_table[3] = 80;
comp_table[4] = 100;
last_comp = 2;
compression_ratio = (UINT32 *) 0x010107D0;

stream_addr[0] = 0x90023000;
stream_addr[1] = 0x90028000;

NOCACHE_INT(number_pixels[0]) = 0x80;

NOCACHE_INT(global_mean) = 0x5e;

*pp_stop_encode = 0;

/* initialize FIRST variable on all PPs - used for first pass # of bitplanes to
process */
tbuf_pp0[0] = 4;
tbuf_pp1[0] = 4;
tbuf_pp2[0] = 4;
tbuf_pp3[0] = 4;

#ifdef HOST
    ReturnVal = CilSigHandler(SignalHandler);
    if (ReturnVal != CIL_OK)
    {
        /** Cannot register signal handler ***/
        while(1);
        return;
    }
#endif

    dx = 512;          /* size of ROI which will be initialised*/
    dy = 480;          /* if too big, may not run at frame rate*/

pVil24 = Vil24Open();          /* Open VIL module          */

```

# NAWCWD TP 8442

```

if ( pVil24 == NULL ) exit(0);          /* Failed to open VIL module */
Vil24Initialize(pVil24,VIL24_EIA_DEFAULT); /* set up for CCIR camera */
Vil24SetVcrMode(pVil24,1);              /* ensure lock to poor sources */
Vil24SetROI(pVil24,64,0,dx,dy);         /* set ROI in the center */

vim = IclCreateHdr(1,1,ICL_IMG_CUSTOM); /* create image header structure ...
*/
Vil24InitImgHdr(pVil24,vim);             /* ... and initialise to describe
VIL */

VolSetDisplay(VOL_VGA8_1024);            /* setup colour display for VIM-8 */
VolSetGreyLUT();

p[0] = (PTREQ *) (MP_PARM_RAM + 0x2c0);
p[1] = p[0] + 1;
p[2] = p[0] + 2;
p[3] = p[0] + 3;
p[4] = p[0] + 4;
p[5] = p[0] + 5;
p[6] = p[0] + 6;
p[7] = p[0] + 7;
p[8] = p[0] + 8;
p[9] = p[0] + 9;

/* Set MP list pointer to point to first PT */
ptr = (long *) (MP_PTREQ_PTR);
*ptr = (long) p[0];

#ifdef INTERLEAVE_CAMERA

/* Fifo Bank 0 -> DRAM (8 bit data)*/

p[0]->link = p[0];                      /* point to next PT */
p[0]->word[0] = 0x80000002;              /* Weird Fifo to linear */
p[0]->word[1] = 0xa0000001;              /* Src address is VIM pixel fifo */
p[0]->word[2] = 0x80300000;              /* Dst address is DRAM */
p[0]->word[3] = 0x01ff0001;              /* Src B count Src A count */
p[0]->word[4] = 0x00000200;              /* Dst B count Dst A count */
p[0]->word[5] = 0x00;                    /* Src C count */
p[0]->word[6] = 0;                       /* Dst C count */
p[0]->word[7] = 0x08;                     /* Src B pitch */
p[0]->word[8] = 0x000;                    /* Dst B pitch */
p[0]->word[9] = 0x400;                     /* Src C pitch */
p[0]->word[10] = 0x200;                   /* Dst C pitch */
p[0]->word[11] = 0;                       /* Src transparency upper */
p[0]->word[12] = 0;                       /* Src transparency lower */
p[0]->word[13] = 0;                       /* Reserved */
p[0]->word[14] = 0;                       /* Reserved */

#else

/* Fifo Bank 0 -> DRAM (8 bit data)*/

p[0]->link = p[0];                      /* point to next PT */
p[0]->word[0] = 0x80000000;              /* Weird Fifo to linear */
p[0]->word[1] = 0xa0000001;              /* Src address is VIM pixel fifo */
p[0]->word[2] = 0x80300000;              /* Dst address is DRAM */
p[0]->word[3] = 0x01ff0001;              /* Src B count Src A count */
p[0]->word[4] = 0x00ef0200;              /* Dst B count Dst A count */
p[0]->word[5] = 0xef;                     /* Src C count */
p[0]->word[6] = 0;                       /* Dst C count */
p[0]->word[7] = 0x08;                     /* Src B pitch */

```

# NAWCWD TP 8442

```

p[0]->word[8] = 0x200;          /* Dst B pitch          */
p[0]->word[9] = 0x400;          /* Src C pitch          */
p[0]->word[10] = 0;             /* Dst C pitch          */
p[0]->word[11] = 0;             /* Src transparency upper */
p[0]->word[12] = 0;             /* Src transparency lower */
p[0]->word[13] = 0;             /* Reserved              */
p[0]->word[14] = 0;             /* Reserved              */

#endif

/* DRAM -> internal coefficient blocks (16 bit data) (32x16 words) */

p[1]->link = p[1];              /* point to next PT      */
p[1]->word[0] = 0x80000000;      /* Contig. Mem to int no update */
p[1]->word[1] = 0x80320000;      /* Src address is DRAM    */
p[1]->word[2] = 0x00008000;      /* Dst address is internal */
p[1]->word[3] = 0x000f0040;      /* Src B count Src A count */
p[1]->word[4] = 0x00000400;      /* Dst B count Dst A count */
p[1]->word[5] = 0x00;           /* Src C count            */
p[1]->word[6] = 0;              /* Dst C count            */
p[1]->word[7] = 0x0400;         /* Src B pitch            */
p[1]->word[8] = 0x000;          /* Dst B pitch            */
p[1]->word[9] = 0x40;           /* Src C pitch            */
p[1]->word[10] = 0x1000;         /* Dst C pitch            */
p[1]->word[11] = 0;             /* Src transparency upper  */
p[1]->word[12] = 0;             /* Src transparency lower  */
p[1]->word[13] = 0;             /* Reserved                */
p[1]->word[14] = 0;             /* Reserved                */

/* DRAM (8 bit data) -> VRAM (raw image) */

p[2]->link = p[2];              /* point to next PT      */
p[2]->word[0] = 0x80000000;      /* linear to VRAM         */
p[2]->word[1] = 0x80300000;      /* Src address is DRAM    */
p[2]->word[2] = 0xb4000000;      /* Dst address is VRAM    */
p[2]->word[3] = 0x00038000;      /* Src B count Src A count */
p[2]->word[4] = 0x00ff0200;      /* Dst B count Dst A count */
p[2]->word[5] = 0x00;           /* Src C count            */
p[2]->word[6] = 0;              /* Dst C count            */
p[2]->word[7] = 0x8000;         /* Src B pitch            */
p[2]->word[8] = 0x800;          /* Dst B pitch            */
p[2]->word[9] = 0x00;           /* Src C pitch            */
p[2]->word[10] = 0x0000;         /* Dst C pitch            */
p[2]->word[11] = 0;             /* Src transparency upper  */
p[2]->word[12] = 0;             /* Src transparency lower  */
p[2]->word[13] = 0;             /* Reserved                */
p[2]->word[14] = 0;             /* Reserved                */

/* DRAM (rows and 8 bit data) -> internal RAM */
/* Can be used 4 times then change dst <- this can be done 8 times */

p[3]->link = p[3];              /* point to next PT      */
p[3]->word[0] = 0x80000202;      /* Contig. Mem to int w/s update */
p[3]->word[1] = 0x80300000;      /* Src address is DRAM    */
p[3]->word[2] = 0x00000000;      /* Dst address is internal */
p[3]->word[3] = 0x00000800;      /* Src B count Src A count */
p[3]->word[4] = 0x00000800;      /* Dst B count Dst A count */
p[3]->word[5] = 0x00;           /* Src C count            */
p[3]->word[6] = 0;              /* Dst C count            */
p[3]->word[7] = 0x0000;         /* Src B pitch            */
p[3]->word[8] = 0x000;          /* Dst B pitch            */
p[3]->word[9] = 0x0800;         /* Src C pitch            */

```



# NAWCWD TP 8442

```

p[3]->word[10] = 0x1000;          /* Dst C pitch          */
p[3]->word[11] = 0;               /* Src transparency upper */
p[3]->word[12] = 0;               /* Src transparency lower */
p[3]->word[13] = 0;               /* Reserved              */
p[3]->word[14] = 0;               /* Reserved              */

/* DRAM (columns and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst <- this can be done 8 times */

p[4]->link = p[4];                /* point to next PT      */
p[4]->word[0] = 0x80000002;        /* Contig. Mem to int w/dst updt */
p[4]->word[1] = 0x80320000;        /* Src address is DRAM    */
p[4]->word[2] = 0x00000000;        /* Dst address is internal */
p[4]->word[3] = 0x00ff0010;        /* Src B count Src A count */
p[4]->word[4] = 0x00001000;        /* Dst B count Dst A count */
p[4]->word[5] = 0x00;              /* Src C count            */
p[4]->word[6] = 0;                 /* Dst C count            */
p[4]->word[7] = 0x0400;            /* Src B pitch            */
p[4]->word[8] = 0x000;             /* Dst B pitch            */
p[4]->word[9] = 0x10;              /* Src C pitch            */
p[4]->word[10] = 0x1000;           /* Dst C pitch            */
p[4]->word[11] = 0;                /* Src transparency upper */
p[4]->word[12] = 0;                /* Src transparency lower */
p[4]->word[13] = 0;                /* Reserved                */
p[4]->word[14] = 0;                /* Reserved                */

/* internal RAM -> DRAM (columns and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[5]->link = p[5];                /* point to next PT      */
p[5]->word[0] = 0x80000200;        /* int to Contig. Mem w/src updt */
p[5]->word[1] = 0x00000000;        /* Src address is internal */
p[5]->word[2] = 0x80320000;        /* Dst address is DRAM    */
p[5]->word[3] = 0x00001000;        /* Src B count Src A count */
p[5]->word[4] = 0x00ff0010;        /* Dst B count Dst A count */
p[5]->word[5] = 0x00;              /* Src C count            */
p[5]->word[6] = 0;                 /* Dst C count            */
p[5]->word[7] = 0x0000;            /* Src B pitch            */
p[5]->word[8] = 0x400;             /* Dst B pitch            */
p[5]->word[9] = 0x1000;            /* Src C pitch            */
p[5]->word[10] = 0x0010;           /* Dst C pitch            */
p[5]->word[11] = 0;                /* Src transparency upper */
p[5]->word[12] = 0;                /* Src transparency lower */
p[5]->word[13] = 0;                /* Reserved                */
p[5]->word[14] = 0;                /* Reserved                */

/* DRAM (rows and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst then this can be repeated 8 times */

p[6]->link = p[6];                /* point to next PT      */
p[6]->word[0] = 0x80000002;        /* Contig. Mem to int w/dst updt */
p[6]->word[1] = 0x80320000;        /* Src address is DRAM    */
p[6]->word[2] = 0x00000000;        /* Dst address is internal */
p[6]->word[3] = 0x00001000;        /* Src B count Src A count */
p[6]->word[4] = 0x00001000;        /* Dst B count Dst A count */
p[6]->word[5] = 0x00;              /* Src C count            */
p[6]->word[6] = 0;                 /* Dst C count            */
p[6]->word[7] = 0x0000;            /* Src B pitch            */
p[6]->word[8] = 0x000;             /* Dst B pitch            */
p[6]->word[9] = 0x0000;            /* Src C pitch            */
p[6]->word[10] = 0x1000;           /* Dst C pitch            */
p[6]->word[11] = 0;                /* Src transparency upper */

```

# NAWCWD TP 8442

```

p[6]->word[12] = 0;          /* Src transparency lower */
p[6]->word[13] = 0;          /* Reserved */
p[6]->word[14] = 0;          /* Reserved */

/* internal RAM -> DRAM (rows and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[7]->link = p[7];          /* point to next PT */
p[7]->word[0] = 0x80000200;   /* int to DRAM w/src updt */
p[7]->word[1] = 0x00000000;   /* Src address is internal */
p[7]->word[2] = 0x80320000;   /* Dst address is DRAM */
p[7]->word[3] = 0x00001000;   /* Src B count Src A count */
p[7]->word[4] = 0x00001000;   /* Dst B count Dst A count */
p[7]->word[5] = 0x00;        /* Src C count */
p[7]->word[6] = 0;           /* Dst C count */
p[7]->word[7] = 0x0000;      /* Src B pitch */
p[7]->word[8] = 0x000;       /* Dst B pitch */
p[7]->word[9] = 0x1000;      /* Src C pitch */
p[7]->word[10] = 0x0000;     /* Dst C pitch */
p[7]->word[11] = 0;          /* Src transparency upper */
p[7]->word[12] = 0;          /* Src transparency lower */
p[7]->word[13] = 0;          /* Reserved */
p[7]->word[14] = 0;          /* Reserved */

/* internal RAM -> VRAM (rows and 8 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[8]->link = p[8];          /* point to next PT */
p[8]->word[0] = 0x80000202;   /* int to VRAM w/dst updt */
p[8]->word[1] = 0x00000000;   /* Src address is internal */
p[8]->word[2] = 0xb4000200;   /* Dst address is VRAM */
p[8]->word[3] = 0x00000800;   /* Src B count Src A count */
p[8]->word[4] = 0x00030200;   /* Dst B count Dst A count */
p[8]->word[5] = 0x00;        /* Src C count */
p[8]->word[6] = 0;           /* Dst C count */
p[8]->word[7] = 0x0000;      /* Src B pitch */
p[8]->word[8] = 0x400;       /* Dst B pitch */
p[8]->word[9] = 0x1000;      /* Src C pitch */
p[8]->word[10] = 0x1000;     /* Dst C pitch */
p[8]->word[11] = 0;          /* Src transparency upper */
p[8]->word[12] = 0;          /* Src transparency lower */
p[8]->word[13] = 0;          /* Reserved */
p[8]->word[14] = 0;          /* Reserved */

/* internal RAM -> DRAM (byte stream 8 bit data) */

p[9]->link = p[9];          /* point to next PT */
p[9]->word[0] = 0x80000202;   /* int to VRAM w/dst updt */
p[9]->word[1] = 0x01000630;   /* Src address is internal */
p[9]->word[2] = 0x80380000;   /* Dst address is VRAM JCWW */

p[9]->word[3] = 0;           /* Src B count Src A count */
p[9]->word[4] = 0;           /* Dst B count Dst A count */
p[9]->word[5] = 0x00;        /* Src C count */
p[9]->word[6] = 0;           /* Dst C count */
p[9]->word[7] = 0x0000;      /* Src B pitch */
p[9]->word[8] = 0x000;       /* Dst B pitch */
p[9]->word[9] = 0x1000;      /* Src C pitch */
p[9]->word[10] = 0;          /* Dst C pitch */
p[9]->word[11] = 0;          /* Src transparency upper */
p[9]->word[12] = 0;          /* Src transparency lower */
p[9]->word[13] = 0;          /* Reserved */

```

# NAWCWD TP 8442

```

        p[9]->word[14] = 0;                                /* Reserved */

        Vil24SetSequenceMode(pVil24,1);                    /* set VIL to sequence mode */
    /*
        Vil24StartCapture(pVil24);                          /* start capturing images */
    */

    /* IE = disable() & 0xfbf0fffe; */

    IE = 0x01;

    command(0x2000000f);                                    /* unhalt PP0 */

    vert_loop_index[0] = 16;
    vert_loop_index[1] = 4;
    vert_loop_index[2] = 1;
    vert_loop_index[3] = 1;
    vert_loop_index[4] = 1;

    horiz_loop_index[0] = 15;
    horiz_loop_index[1] = 5;
    horiz_loop_index[2] = 1;
    horiz_loop_index[3] = 1;
    horiz_loop_index[4] = 1;

    jump_col[0] = 16;
    jump_col[1] = 64;
    jump_col[2] = 256;
    jump_col[3] = 256;
    jump_col[4] = 256;

    jump_row[0] = 0x1000; /* not used */
    jump_row[1] = 0x3000;
    jump_row[2] = 0xF000;
    jump_row[3] = 0x10000;
    jump_row[4] = 0x10000;

    /* Zero out unused lines in the input (lines 240-255) */

    /* unused_pixels = (long *) 0x8031e000;

    for (i=0;i<2048;i++)
        NOCACHE_INT(unused_pixels[i]) = 0; */

    /* setup for fill with value PT */

        p[9]->link = p[9];                                /* point to next PT */
        p[9]->word[0] = 0x80001000;                        /* int to VRAM w/dst updt */
        p[9]->word[1] = 0x00000000;                        /* Src address is internal */
        p[9]->word[2] = 0x8031e000;                        /* Dst address is VRAM JCWW */
    /*
        p[9]->word[3] = 0x00000000;                        /* Src B count Src A count */
        p[9]->word[4] = 0x00002000;                        /* Dst B count Dst A count */
        p[9]->word[5] = 0x00;                              /* Src C count */
        p[9]->word[6] = 0;                                  /* Dst C count */
        p[9]->word[7] = 0x0000;                            /* Src B pitch */
        p[9]->word[8] = 0x0000;                            /* Dst B pitch */
        p[9]->word[9] = 0x0000;                            /* Src C pitch */
        p[9]->word[10] = 0;                                 /* Dst C pitch */
        p[9]->word[11] = 0;                                 /* Src transparency upper */
        p[9]->word[12] = 0;                                 /* Src transparency lower */
        p[9]->word[13] = 0;                                 /* Reserved */
    */

```

# NAWCWD TP 8442

```

    p[9]->word[14] = 0;                /* Reserved */

/* zero out unused pixels using a fill with value packet transfer */
    *ptr = (long) p[9];

/* kick off TC to transfer pixel info to DRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

    p[9]->word[2] = 0x80420000;          /* Dst address is VRAM JCWW */
/*
    p[9]->word[4] = 0x003B1000;          /* Dst B count Dst A count */
    p[9]->word[8] = 0x1000;             /* Dst B pitch */

/* zero out buffer */
    *ptr = (long) p[9];

/* kick off TC to transfer pixel info to DRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

/* re-initialize p[9] */

    p[9]->link = p[9];                  /* point to next PT */
    p[9]->word[0] = 0x80000202;          /* int to VRAM w/dst updt */
    p[9]->word[1] = 0x01000630;          /* Src address is internal */
    p[9]->word[2] = 0x80380000;          /* Dst address is VRAM JCWW */
/*
    p[9]->word[3] = 0;                  /* Src B count Src A count */
    p[9]->word[4] = 0;                  /* Dst B count Dst A count */
    p[9]->word[5] = 0x00;               /* Src C count */
    p[9]->word[6] = 0;                  /* Dst C count */
    p[9]->word[7] = 0x0000;             /* Src B pitch */
    p[9]->word[8] = 0x000;             /* Dst B pitch */
    p[9]->word[9] = 0x1000;             /* Src C pitch */
    p[9]->word[10] = 0;                 /* Dst C pitch */
    p[9]->word[11] = 0;                 /* Src transparency upper */
    p[9]->word[12] = 0;                 /* Src transparency lower */
    p[9]->word[13] = 0;                 /* Reserved */
    p[9]->word[14] = 0;                 /* Reserved */

```

# NAWCWD TP 8442

```

/* Initialize the adaptive bit allocation tables on the PPs */

    init_alloc_table(2);

    Vil24SetReadBank(pVil24,0);          /* prepare to read 1st field
*/

/*****
*/

Vil24WaitForData(pVil24);                /* wait for image data          */
TCOUNT = 0xffffffff;

channel_bandwidth = 48000.0;
bits_per_frame = (YSIZE*XSIZE/comp_table[last_comp] - headersize) * 8.0;
fps = channel_bandwidth / bits_per_frame;
secs_between_frames = 1.0/fps;
ticks_between_frames = secs_between_frames/0.000000025;
time_wait = ticks_between_frames;
time_wait = 0xffffffff - time_wait;

    while (1)
    {

#ifdef CHANNEL_LIMIT
        while (TCOUNT > time_wait);
#endif

#ifdef CAMERA
        Vil24WaitForData(pVil24);          /* wait for image data
*/
        Vil24SetReadBank(pVil24,0);      /* prepare to read 1st field
*/
#endif

        encode_time = 0xffffffff - TCOUNT;
        TCOUNT = 0xffffffff;

#ifdef CAMERA
        *ptr = (long) p[0];

/* kick off TC to transfer pixel info to DRAM */
        PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);
#endif

/* clear the interrupt flag */
        INTPEN = 0xf0000;

#ifdef HOST

#ifdef variable_compression
/* Read Compression Ratio */

CilReadMailbox(1,(PUINT32)compression_ratio);

channel_bandwidth = ((*compression_ratio) >> 8) * 1.0;

if (channel_bandwidth < 10) channel_bandwidth = 56000.0;

if (channel_bandwidth > 100000001) channel_bandwidth = 56000.0;

*compression_ratio = *compression_ratio & 0xff;

```

# NAWCWD TP 8442

```

if (*compression_ratio > 4) *compression_ratio = 2;
if (*compression_ratio < 0) *compression_ratio = 2;

if (*compression_ratio != last_comp)
{
    last_comp = *compression_ratio;
    init_alloc_table(last_comp);
}

bits_per_frame = (YSIZE*XSIZE/comp_table[last_comp] - headersize) * 8.0;
fps = channel_bandwidth / bits_per_frame;
secs_between_frames = 1.0/fps;
ticks_between_frames = secs_between_frames/0.000000025;
time_wait = ticks_between_frames;
time_wait = 0xffffffff - time_wait;

#endif

#ifdef variable_compression
/* Read Compression Ratio */

CilReadMailbox(1, (PUINT32)compression_ratio);

if (*compression_ratio > 16640) *compression_ratio = 16640;
if (*compression_ratio < 480) *compression_ratio = 480;

if (*compression_ratio != last_comp)
{
    last_comp = *compression_ratio;
    init_alloc_table2(last_comp);
}
#endif

#endif

/* forget second field */

/* Display the incoming raw data */

p[2]->word[2] = 0xb4000000;

*ptr = (long) p[2];

/* kick off TC to transfer pixel from DRAM to VRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

p[2]->word[2] = 0xb4000400;

/* kick off TC to transfer pixel from DRAM to VRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = 0; while(PKTREQ & 0x02);

/* Encode coefficients of the raw image */

#ifdef ENCODE

/* TCOUNT = 0xffffffff; */

p[3]->word[1] = 0x80300000; /* Src address is DRAM */

for (lp=0; lp<5; lp++)
{

```

```

/* Do 32 rows */

for (i=0; i<horiz_loop_index[lp]; i++)
{
    if (i==0)
    {
        p[6]->word[1] = 0x80320000;          /* Src address is SDRAM      */
        p[7]->word[2] = 0x80320000;          /* Dst address is SDRAM     */

        p[3]->word[2] = 0x00000000;          /* Dst address is internal  */
        p[6]->word[2] = 0x00000000;          /* Dst address is internal  */
        p[7]->word[1] = 0x00000000;          /* Src address is internal  */

        if (lp==0)
            *ptr = (long) p[3];
        else
            *ptr = (long) p[6];

        if (i==0)
        {
            switch (lp) {
                case 0:
                {
/* p[6] never used here to move coefficients in the first time, p[3] is used
instead */
                    p[6]->word[3] = 0x00001000;
                    p[6]->word[4] = 0x00001000;
                    p[6]->word[5] = 0x00000000;
                    p[6]->word[7] = 0x00000000;
                    p[6]->word[9] = 0x00000000;

                    p[7]->word[3] = 0x00001000;
                    p[7]->word[4] = 0x00001000;
                    p[7]->word[6] = 0x00000000;
                    p[7]->word[8] = 0x00000000;
                    p[7]->word[10] = 0x00000000;
                    break;
                }
                case 1:
                {
                    p[6]->word[3] = 0x00ff0002;
                    p[6]->word[4] = 0x00000c00;
                    p[6]->word[5] = 0x00000005;
                    p[6]->word[7] = 0x00000004;
                    p[6]->word[9] = 0x00000800;

                    p[7]->word[3] = 0x00000c00;
                    p[7]->word[4] = 0x00ff0002;
                    p[7]->word[6] = 0x00000005;
                    p[7]->word[8] = 0x00000004;
                    p[7]->word[10] = 0x00000800;
                    break;
                }
                case 2:
                {
                    p[6]->word[3] = 0x007f0002;
                    p[6]->word[4] = 0x00000f00;
                    p[6]->word[5] = 0x0000000e;
                    p[6]->word[7] = 0x00000008;
                }
            }
        }
    }
}

```

```

    p[6]->word[9] = 0x00001000;

    p[7]->word[3] = 0x00000f00;
    p[7]->word[4] = 0x007f0002;
    p[7]->word[6] = 0x0000000e;
    p[7]->word[8] = 0x00000008;
    p[7]->word[10] = 0x00001000;
    break;
}
case 3:
{
    p[6]->word[3] = 0x003f0002;
    p[6]->word[4] = 0x00000400;
    p[6]->word[5] = 0x00000007;
    p[6]->word[7] = 0x00000010;
    p[6]->word[9] = 0x00002000;

    p[7]->word[3] = 0x00000400;
    p[7]->word[4] = 0x003f0002;
    p[7]->word[6] = 0x00000007;
    p[7]->word[8] = 0x00000010;
    p[7]->word[10] = 0x00002000;
    break;
}
case 4:
{
    p[6]->word[3] = 0x001f0002;
    p[6]->word[4] = 0x00000100;
    p[6]->word[5] = 0x00000003;
    p[6]->word[7] = 0x00000020;
    p[6]->word[9] = 0x00004000;

    p[7]->word[3] = 0x00000100;
    p[7]->word[4] = 0x001f0002;
    p[7]->word[6] = 0x00000003;
    p[7]->word[8] = 0x00000020;
    p[7]->word[10] = 0x00004000;
    break;
}
default:
    break;
}
}

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002001); /* send msg interrupt to PP0 */
#endif

p[6]->word[1] = p[6]->word[1] + jump_row[lp];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

```



# NAWCWD TP 8442

```

    while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002002);          /* send msg interrupt to PP1 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002004);          /* send msg interrupt to PP2 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002008);          /* send msg interrupt to PP3 */
#endif

    p[6]->word[1] = p[6]->word[1] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x10000)==0x00); /* poll PP0 */
    INTPEN = 0x10000;                /* clear the interrupt flag */
#endif

    *ptr = (long) p[7];

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x20000)==0x00); /* poll PP1 */
    INTPEN = 0x20000;                /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

```

```

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x40000)==0x00);    /* poll PP2 */
    INTPEN = 0x40000;                  /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

#ifdef DEBUG
    while((INTPEN & 0x80000)==0x00);    /* poll PP3 */
    INTPEN = 0x80000;                  /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[7]->word[2] = p[7]->word[2] + jump_row[lp];

    } /* end for i */

NOCACHE_INT(global_mean) = (NOCACHE_INT(mean_pp[0]) + NOCACHE_INT(mean_pp[1]) +
NOCACHE_INT(mean_pp[2]) + NOCACHE_INT(mean_pp[3])) / (512*240);

/* Do the appropriate number of columns depending on the scale */

if (lp!=4)
    for (i=0; i<vert_loop_index[lp]; i++)
    {
        if (i==0)
        {
            p[4]->word[1] = 0x80320000;    /* Src address is DRAM */
            p[5]->word[2] = 0x80320000;    /* Dst address is DRAM */
        }

        p[4]->word[2] = 0x00000000;    /* Dst address is internal */

        *ptr = (long) p[4];
    }

```

```

#ifdef DEBUG
/* JCWtest */
if (i==1)
{
    for (junki=0;junki<256*256;junki=junki+1)
    {
        fillme[junki] = junkfill;
        junkfill = junkfill + 1;
    }
}
#endif

if (i==0)
{
    switch (lp) {
        case 0:
        {
            p[4]->word[3] = 0x00ef0010;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x00000000;
            p[4]->word[7] = 0x00000400;
            p[4]->word[9] = 0x00000000;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x00ef0010;
            p[5]->word[6] = 0x00000000;
            p[5]->word[8] = 0x00000400;
            p[5]->word[10] = 0x00000000;
            break;
        }
        case 1:
        {
            p[4]->word[3] = 0x000f0002;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x00000077;
            p[4]->word[7] = 0x00000004;
            p[4]->word[9] = 0x00000800;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x000f0002;
            p[5]->word[6] = 0x00000077;
            p[5]->word[8] = 0x00000004;
            p[5]->word[10] = 0x00000800;
            break;
        }
        case 2:
        {
            p[4]->word[3] = 0x001f0002;
            p[4]->word[4] = 0x00000f00;
            p[4]->word[5] = 0x0000003b;
            p[4]->word[7] = 0x00000008;
            p[4]->word[9] = 0x00001000;

            p[5]->word[3] = 0x00000f00;
            p[5]->word[4] = 0x001f0002;
            p[5]->word[6] = 0x0000003b;
            p[5]->word[8] = 0x00000008;
            p[5]->word[10] = 0x00001000;
            break;
        }
    }
}

```

```

    case 3:
    {
        p[4]->word[3] = 0x000f0002;
        p[4]->word[4] = 0x000003c0;
        p[4]->word[5] = 0x0000001d;
        p[4]->word[7] = 0x00000010;
        p[4]->word[9] = 0x00002000;

        p[5]->word[3] = 0x000003c0;
        p[5]->word[4] = 0x000f0002;
        p[5]->word[6] = 0x0000001d;
        p[5]->word[8] = 0x00000010;
        p[5]->word[10] = 0x00002000;
        break;
    }
    default:
        break;
    }
}

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002001);          /* send msg interrupt to PP0 */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp];          /* Src address is DRAM */
/*

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002002);          /* send msg interrupt to PP1 */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM */
/*

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002004);          /* send msg interrupt to PP2 */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM */
/*

```

# NAWCWD TP 8442

```

/* kick off TC to transfer columns of pixels from DRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT;

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

#ifdef DEBUG
    command(0x00002008);          /* send msg interrupt to PP3 */
#endif

p[4]->word[1] = p[4]->word[1] + jump_col[lp]; /* Src address is DRAM */

p[5]->word[1] = 0x00000000;      /* Src address is internal */
*ptr = (long) p[5];

#ifdef DEBUG
    while((INTPEN & 0x10000)==0x00); /* poll PP0 */
    INTPEN = 0x10000;              /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */
*/

#ifdef DEBUG
    while((INTPEN & 0x20000)==0x00); /* poll PP1 */
    INTPEN = 0x20000;              /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */
*/

#ifdef DEBUG
    while((INTPEN & 0x40000)==0x00); /* poll PP2 */
    INTPEN = 0x40000;              /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM */
*/

```

# NAWCWD TP 8442

```

#ifdef DEBUG
    while((INTPEN & 0x80000)==0x00); /* poll PP3 */
    INTPEN = 0x80000; /* clear the interrupt flag */
#endif

    PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from internal to DRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

    p[5]->word[2] = p[5]->word[2] + jump_col[lp]; /* Dst address is DRAM
*/

}

} /* end for lp */

#ifdef OLD_DISPLAY
/* Display the coefficients */

    p[6]->word[1] = 0x80320000; /* Src address is DRAM */

    p[6]->word[3] = 0x00001000;
    p[6]->word[4] = 0x00001000;
    p[6]->word[5] = 0x00000000;
    p[6]->word[7] = 0x00000000;
    p[6]->word[9] = 0x00001000;

    p[8]->word[2] = 0xb4080000; /* Dst address is VRAM */

    for (i=0;i<16;i++)
    {
        *ptr = (long) p[6];

        p[6]->word[2] = 0x00000000; /* Dst address is internal */

        PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from DRAM to internal */

        i = i + 1;
        i = i - 1;

        while(PKTREQ & 0x02);

        command(0x00002001); /* send msg interrupt to PP0 */

        p[6]->word[1] = p[6]->word[1] + jump_row[0];

        PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

        i = i + 1;
        i = i - 1;

        while(PKTREQ & 0x02);

        command(0x00002002); /* send msg interrupt to PP1 */
    }

```

# NAWCWD TP 8442

```

p[6]->word[1] = p[6]->word[1] + jump_row[0];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002004);          /* send msg interrupt to PP2      */

p[6]->word[1] = p[6]->word[1] + jump_row[0];

PKTREQ |= MP_PKTREQ_P_BIT; /* kick off transfer from SDRAM to internal */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

command(0x00002008);          /* send msg interrupt to PP3      */

p[6]->word[1] = p[6]->word[1] + jump_row[0];

while((INTPEN & 0x10000)==0x00); /* poll PP0 */
INTPEN = 0x10000;               /* clear the interrupt flag */

*ptr = (long) p[8];

p[8]->word[1] = 0x00000000;      /* Src address is internal      */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

while((INTPEN & 0x20000)==0x00); /* poll PP1 */
INTPEN = 0x20000;               /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

while((INTPEN & 0x40000)==0x00); /* poll PP2 */
INTPEN = 0x40000;               /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

i = i + 1;
i = i - 1;

while(PKTREQ & 0x02);

while((INTPEN & 0x80000)==0x00); /* poll PP3 */
INTPEN = 0x80000;               /* clear the interrupt flag */

PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

```

```

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);

}

#endif

#endif

*pp_stop_encode = 0;

#ifdef ENCODE_STREAM

/*****

#ifdef HOST
    /* Tell host where bit stream can be found */

    ReturnVal = CilWriteMailbox(0,stream_addr[flip]);

    if (ReturnVal != CIL_OK)
    {
        /*** Cannot write to mailbox ***/
        while(1);
        return;
    }

#endif

/* DRAM (rows and 8 bit data) -> internal RAM */
/* Can be used 4 times then change dst <- this can be done 8 times */

    p[3]->word[0] = 0x80000002;          /* Contig. Mem to int w/s update */
    p[3]->word[1] = 0x00000000;          /* Src address is DRAM */
    p[3]->word[2] = 0x80420000;          /* Dst address is internal */
    p[3]->word[3] = 0x00000C00;          /* Src B count Src A count */
    p[3]->word[4] = 0x00000C00;          /* Dst B count Dst A count */
    p[3]->word[5] = 0x00;                /* Src C count */
    p[3]->word[6] = 0;                  /* Dst C count */
    p[3]->word[7] = 0x0000;             /* Src B pitch */
    p[3]->word[8] = 0x000;              /* Dst B pitch */
    p[3]->word[9] = 0x1000;             /* Src C pitch */
    p[3]->word[10] = 0x0C00;            /* Dst C pitch */

/* internal RAM -> DRAM (rows and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

    p[7]->word[0] = 0x80000202;          /* int to DRAM w/dst updt */
    p[7]->word[1] = 0x00000000;          /* Dst address is SDRAM */
    p[7]->word[2] = 0x80400000;          /* Dst address is SDRAM */
    p[7]->word[3] = 0x00000c00;          /* Src B count Src A count */
    p[7]->word[4] = 0x00000c00;          /* Dst B count Dst A count */
    p[7]->word[6] = 0x00000000;
    p[7]->word[8] = 0x00000000;
    p[7]->word[9] = 0x1000;              /* Src C pitch */
    p[7]->word[10] = 0x0c00;            /* Dst C pitch */

```



# NAWCWD TP 8442

```

    p[9]->word[2] = stream_addr[flip] + 8;          /* Dst address is
external SDRAM */

/* DRAM (rows and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst then this can be repeated 8 times */

    p[6]->word[0] = 0x80000200;          /* Contig. Mem to int w/dst updt */
    p[6]->word[1] = 0x80420000;          /* Src address is DRAM */
    p[6]->word[2] = 0x00000000;          /* Dst address is internal */
    p[6]->word[3] = 0x00000400;          /* Src B count Src A count */
    p[6]->word[4] = 0x00000400;          /* Dst B count Dst A count */
    p[6]->word[5] = 0x00;                /* Src C count */
    p[6]->word[6] = 0;                    /* Dst C count */
    p[6]->word[7] = 0x0000;              /* Src B pitch */
    p[6]->word[8] = 0x000;                /* Dst B pitch */
    p[6]->word[9] = 0x0400;              /* Src C pitch */
    p[6]->word[10] = 0x0000;             /* Dst C pitch */

    for (current_block=0;current_block<40;current_block = current_block + 4)
    {
        for (j=0;j<3;j++)
        {
            for (i=0;i<4;i++)
            {
                if (j!=0)
                {
                    while((INTPEN & (1<<(16+i)))==0x00); /* poll PP for finished block
transformations */
                    INTPEN = 0x10000<<i; /* clear PP interrrupt */
                }

                p[1]->word[1] = 0x80320000 + ((current_block+i)>>3)*0x04000 +
((current_block+i)&7)*64 + 0x14000*j; /* Src for block changes */
                p[1]->word[2] = 0x00008000 + (i<<12); /* Dst address is
internal RAM2 */

                *ptr = (long) p[1];

                /* kick off 1024 byte (32x16 words) coefficient block transfer from
SDRAM to internal */
                PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

                p[1]->word[1] = p[1]->word[1] + 0x200; /* Src for block
changes */
                p[1]->word[2] = p[1]->word[2] + 0x400; /* Dst address is
internal DATA RAM2 */

                /* kick off 1024 byte (32x16 words) coefficient block transfer from
SDRAM to internal */
                PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

                command(0x00002000 + (1<<i)); /* send msg interrupt to PP that will
process these 2 blocks */
            } /* end for i */
        } /* end for j */

        /* Move Buffer (0x80420000) to internal memory (2 parts) */

        for (i=0;i<4;i++)
        {

```

# NAWCWD TP 8442

```

        while((INTPEN & (1<<(16+i)))==0x00);    /* poll PP for finished block
transformations */
        INTPEN = 0x10000<<i;                    /* clear PP interrrupt */

        /* Move buffer in (two steps required) */

        p[6]->word[2] = 0xc00 + i*0x1000;        /*Dst is DATA RAM 1 (half way
thru) */
        p[6]->word[3] = p[6]->word[4] = p[6]->word[9] = 0x400;    /* length is 1024
bytes */
        *ptr = (long) p[6];

        /* kick off 1024 byte (2x256x1 words) coefficient block transfer from
SDRAM to internal */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        p[6]->word[2] = 0x8000 + i*0x1000;        /* Dst is DATA RAM 2 */
        p[6]->word[3] = p[6]->word[4] = p[6]->word[9] = 0x800;    /* length is 2048
bytes */

        /* kick off 2048 byte (4x256x1 words) coefficient block transfer from
SDRAM to internal */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        command(0x00002000 + (1<<i));    /* send msg interrupt to PP to difference
values and start Quantization */
    }

    /* Move internal memory to buffer */

    p[7]->word[1] = 0x0000;                    /* Src for block changes */

    *ptr = (long) p[7];

    for (i=0;i<4;i++)
    {
        while((INTPEN & (1<<(16+i)))==0x00);    /* poll PP for finished block
transformations */
        INTPEN = 0x10000<<i;                    /* clear PP interrrupt */

        /* kick off 3072 byte (6x256x1 words) coefficient block transfer from
internal to 0x80400000 */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
    }

    } /* end for current_block */

/* Read back local maxvals and compute global maxval for all PP's */

temp_maxval = NOCACHE_INT(local_maxval[0]);

if (NOCACHE_INT(local_maxval[1]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[1]);
if (NOCACHE_INT(local_maxval[2]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[2]);
if (NOCACHE_INT(local_maxval[3]) > temp_maxval) temp_maxval =
NOCACHE_INT(local_maxval[3]);

i=0;
while(temp_maxval!=1)
{

```

# NAWCWD TP 8442

```

    i++;
    temp_maxval = temp_maxval>>1;
}
temp_maxval = 1<<i;

NOCACHE_INT(global_maxval) = i;

command(0x0000200F);          /* send msg interrupt to PP0,1,2,3 */

/* Continue with Quantization and differencing */

p[6]->word[1] = 0x80420000;
p[3]->word[2] = 0x80420000;          /* Src for block changes */

p[7]->word[1] = 0x80400000;
p[7]->word[10] = 0x0000;

for (current_block=0;current_block<40;current_block = current_block + 4)
{

/* Write coefficients back into internal memory (0x0000) from external buffer
(0x80400000) and kick off Quant. */

    save_p7_src = p[7]->word[1];

    for (i=0;i<4;i++)
    {

        p[7]->word[2] = i*0x1000;          /*Dst is DATA RAM 1 (half way thru) */
        p[7]->word[3] = p[7]->word[4] = p[7]->word[9] = 0xc00; /* length is 3072
bytes */
        *ptr = (long) p[7];

        /* kick off 1024 byte (2x256x1 words) coefficient block transfer from
SDRAM to internal */
        PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

        *(unsigned short *) (0x01000600 + (i<<12)) = local_alloc[current_block+i];

        command(0x00002000 + (1<<i)); /* send msg interrupt to PP to start
Quantization */
    }

    p[9]->word[1] = 0x01000630;          /* Src address is internal Parameter RAM
*/

    p[7]->word[1] = save_p7_src;

    for (i=0;i<4;i++)
    {
        /* Read the number of bytes that need to be transferred */

        p[9]->word[3] = p[9]->word[4] = p[9]->word[10] =
local_alloc[current_block+i];
        *ptr = (long) p[9];

        while((INTPEN & (1<<(16+i)))==0x00); /* poll PP for finished
Quantization */
        INTPEN = 0x10000<<i;          /* clear PP interrupt */
    }

```

# NAWCWD TP 8442

```

/* kick off transfer from internal to SDRAM */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

/* Move coefficients in (two steps required) */

thru) p[7]->word[2] = 0xc00 + i*0x1000;          /*Dst is DATA RAM 1 (half way
*/
p[7]->word[3] = p[7]->word[4] = p[7]->word[9] = 0x400; /* length is 1024
bytes */
*ptr = (long) p[7];

/* kick off 1024 byte (2x256x1 words) coefficient block transfer from
SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

p[7]->word[2] = 0x8000 + i*0x1000;          /* Dst is DATA RAM 2 */
p[7]->word[3] = p[7]->word[4] = p[7]->word[9] = 0x800; /* length is 2048
bytes */

/* kick off 2048 byte (4x256x1 words) coefficient block transfer from
SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

command(0x00002000 + (1<<i)); /* send msg interrupt to PP to special
difference values */
}

/* Move buffer to internal memory (2 parts) */

for (i=0;i<4;i++)
{
while((INTPEN & (1<<(16+i)))==0x00); /* poll PP for finished block
transformations */
INTPEN = 0x10000<<i;          /* clear PP interrrupt */

/* Move coefficients from buffer in */

p[6]->word[2] = i*0x1000 + 0xc00;          /*Dst is DATA RAM 1 */
p[6]->word[3] = p[6]->word[4] = p[6]->word[9] = 0x400; /* length is 1024
bytes */
*ptr = (long) p[6];

/* kick off 1024 byte (2x256x1 words) coefficient block transfer from
SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

p[6]->word[2] = i*0x1000 + 0x8000;          /*Dst is DATA RAM 2 */
p[6]->word[3] = p[6]->word[4] = p[6]->word[9] = 0x800; /* length is 2048
bytes */
*ptr = (long) p[6];

/* kick off 1024 byte (2x256x1 words) coefficient block transfer from
SDRAM to internal */
PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);

command(0x00002000 + (1<<i)); /* send msg interrupt to PP to sum values
*/
}

/* Move internal memory to buffer */

*ptr = (long) p[3];

```

# NAWCWD TP 8442

```

for (i=0;i<4;i++)
{
    while((INTPEN & (1<<(16+i)))==0x00);    /* poll PP for finished block
transformations */
    INTPEN = 0x10000<<i;                    /* clear PP interrupt */

    p[3]->word[1] = i*0x1000;                /* Src for block changes */

    /* kick off 3072 byte (6x256x1 words) coefficient block transfer from
internal to SDRAM */
    PKTREQ |= MP_PKTREQ_P_BIT; i = i + 1; i = i - 1; while(PKTREQ & 0x02);
}

}

/* internal RAM -> DRAM (rows and 16 bit data) */
/* Can be used 4 times then change src then this can be repeated 8 times */

p[7]->word[0] = 0x80000200;                /* int to DRAM w/src updt */
p[7]->word[1] = 0x00000000;                /* Src address is internal */
p[7]->word[2] = 0x80320000;                /* Dst address is DRAM */
p[7]->word[3] = 0x00001000;                /* Src B count Src A count */
p[7]->word[4] = 0x00001000;                /* Dst B count Dst A count */
p[7]->word[9] = 0x1000;                    /* Src C pitch */
p[7]->word[10] = 0x0000;                   /* Dst C pitch */

/* DRAM (rows and 16 bit data) -> internal RAM */
/* Can be used 4 times then change dst then this can be repeated 8 times */

p[6]->word[0] = 0x80000002;                /* Contig. Mem to int w/dst updt */
p[6]->word[1] = 0x80320000;                /* Src address is DRAM */
p[6]->word[2] = 0x00000000;                /* Dst address is internal */
p[6]->word[3] = 0x00001000;                /* Src B count Src A count */
p[6]->word[4] = 0x00001000;                /* Dst B count Dst A count */
p[6]->word[5] = 0x00;                      /* Src C count */
p[6]->word[6] = 0;                          /* Dst C count */
p[6]->word[7] = 0x0000;                    /* Src B pitch */
p[6]->word[8] = 0x000;                     /* Dst B pitch */
p[6]->word[9] = 0x0000;                    /* Src C pitch */
p[6]->word[10] = 0x1000;                   /* Dst C pitch */

/* DRAM (rows and 8 bit data) -> internal RAM */
/* Can be used 4 times then change dst <- this can be done 8 times */

p[3]->word[0] = 0x80000202;                /* Contig. Mem to int w/s update */
p[3]->word[1] = 0x80300000;                /* Src address is DRAM */
p[3]->word[2] = 0x00000000;                /* Dst address is internal */
p[3]->word[3] = 0x00000800;                /* Src B count Src A count */
p[3]->word[4] = 0x00000800;                /* Dst B count Dst A count */
p[3]->word[5] = 0x00;                      /* Src C count */
p[3]->word[6] = 0;                          /* Dst C count */
p[3]->word[7] = 0x0000;                    /* Src B pitch */
p[3]->word[8] = 0x000;                     /* Dst B pitch */
p[3]->word[9] = 0x0800;                    /* Src C pitch */
p[3]->word[10] = 0x1000;                   /* Dst C pitch */

```

# NAWCWD TP 8442

```

ppstarttime = TCOUNT;

*pp_stop_encode = 1;
command(0x0000200F);          /* send msg interrupt to all PPs */

#endif

/* Write dynamic values to the bitstream */

temp_ulong = 0;
temp_ulong = temp_ulong | (tbuf_pp0[0] << 24);
temp_ulong = temp_ulong | (global_mean << 16);
temp_ulong = temp_ulong | (global_maxval << 8);

NOCACHE_INT(* (UINT32 *) stream_addr[flip]) = temp_ulong;

temp_ulong = *compression_ratio;

#ifdef variable_compression
    NOCACHE_INT(* (UINT32 *) (stream_addr[flip] + 4)) = temp_ulong & 0x000000ff;
#else
    NOCACHE_INT(* (UINT32 *) (stream_addr[flip] + 4)) = temp_ulong;
#endif

flip = 1 - flip;

#ifdef HOST
    ReturnVal = CilRaiseSignalNumber(FrameDoneBit);
    if (ReturnVal != CIL_OK)
    {
        /*** Cannot raise signal ***/
        return;
    }
#endif

proc_time = 0xffffffff - TCOUNT;

#ifdef SHOWDISPLAY

if (Semaphore == 0)
{
    p[7]->word[3] = 0x00001000;
    p[7]->word[4] = 0x00001000;
    p[7]->word[6] = 0x00000000;
    p[7]->word[8] = 0x00000000;
    p[7]->word[10] = 0x00001000;

    /* DRAM (8 bit data) -> VRAM (raw image) */

    p[2]->link = p[2];          /* point to next PT */
    p[2]->word[0] = 0x80000000;  /* linear to VRAM */
    p[2]->word[1] = 0x80300000;  /* Src address is DRAM */
    p[2]->word[2] = 0xb4000000;  /* Dst address is VRAM */
    p[2]->word[3] = 0x00110010;  /* Src B count Src A count */
    p[2]->word[4] = 0x00110010;  /* Dst B count Dst A count */
    p[2]->word[5] = 0x00;        /* Src C count */
    p[2]->word[6] = 0;           /* Dst C count */
}

```

# NAWCWD TP 8442

```

    p[2]->word[7] = 0xb0;          /* Src B pitch          */
    p[2]->word[8] = 0x400;         /* Dst B pitch          */
    p[2]->word[9] = 0x00;         /* Src C pitch          */
    p[2]->word[10] = 0x0000;       /* Dst C pitch          */
    p[2]->word[11] = 0;           /* Src transparency upper */
    p[2]->word[12] = 0;           /* Src transparency lower */
    p[2]->word[13] = 0;           /* Reserved             */
    p[2]->word[14] = 0;           /* Reserved             */

temp_time = 1.0/(encode_time*0.000000025);
/* temp_time = encode_time*0.000025; */

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
    else
    {
        temp_str = 160;
    }

    p[2]->word[1] = (long)&number_pixels;
    p[2]->word[1] += temp_str;
    p[2]->word[2] = 0xb4080000 + i*16;

    *ptr = (long) p[2];

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);
}

temp_time = proc_time*0.000025;

sprintf(string1,"%f",temp_time);

for (i=0;i<4;i++)
{
    if (string1[i]!='.')
    {
        temp_str = string1[i];
        temp_str = temp_str - 48;
        temp_str = temp_str * 16;
    }
    else
    {
        temp_str = 160;
    }

    p[2]->word[1] = (long)&number_pixels;
    p[2]->word[1] += temp_str;
    p[2]->word[2] = 0xb4084800 + i*16;

```

```

    *ptr = (long) p[2];

    PKTREQ |= MP_PKTREQ_P_BIT; /* start xfer from internal to VRAM */

    i = i + 1;
    i = i - 1;

    while(PKTREQ & 0x02);
}

/* DRAM (8 bit data) -> VRAM (raw image) */

    p[2]->link = p[2];                /* point to next PT          */
    p[2]->word[0] = 0x80000000;        /* linear to VRAM          */
    p[2]->word[1] = 0x80300000;        /* Src address is DRAM     */
    p[2]->word[2] = 0xb4000000;        /* Dst address is VRAM    */
    p[2]->word[3] = 0x00038000;        /* Src B count Src A count */
    p[2]->word[4] = 0x00ff0200;        /* Dst B count Dst A count */
    p[2]->word[5] = 0x00;              /* Src C count             */
    p[2]->word[6] = 0;                 /* Dst C count             */
    p[2]->word[7] = 0x8000;            /* Src B pitch             */
    p[2]->word[8] = 0x800;             /* Dst B pitch             */
    p[2]->word[9] = 0x00;              /* Src C pitch             */
    p[2]->word[10] = 0x0000;           /* Dst C pitch             */
    p[2]->word[11] = 0;                /* Src transparency upper  */
    p[2]->word[12] = 0;                /* Src transparency lower  */
    p[2]->word[13] = 0;                /* Reserved                */
    p[2]->word[14] = 0;                /* Reserved                */

}

#endif

} /* end while */

} /* end task */

extern int ep_runpp0;

main()
{
    int i;
    unsigned int temp;
    unsigned int *src_ptr = (unsigned int *)0x90080000;
    unsigned int *dst_ptr = (unsigned int *)0x80000000;

    /* REFCNTL = 0xffff0138; */ /* setup up dram and sdram to correct refresh rate
    for 40 Mhz C80*/
    REFCNTL = 0xffff0186; /* setup up dram and sdram to correct refresh rate for 50
    Mhz C80*/

    command(0xc000000f);                /* reset and halt PP0,1,2,3 */

    *(int *)0x010001b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010011b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010021b8 = (int)&ep_runpp0; /* initialize task vector */
    *(int *)0x010031b8 = (int)&ep_runpp0; /* initialize task vector */

    /* upload PP code */

    /* memcpy( 0x80000000, 0x90200000, 0x9020bae0 - 0x90200000); */

```



# NAWCWD TP 8442

```

for (i=0;i<34000;i++)
{
    temp = *src_ptr++;
    *dst_ptr++ = temp;
}

for (i=0;i<20000;i++)
{
    temp = temp + 1;
    temp = temp -1;
}

command(0x3000000f);          /* start PP0,1,2,3 by unhalting it */
                             /* all will take its task interrupt*/

/* Basic init functions */

#ifdef HOST
    InterruptInit();          /* Init ME interrupts */
#endif

/* Basic init functions */

PtReqInit();                 /* Init the ME PT functions */
TaskInitTasking();           /* Init tasking */
IclInstallPtdMalloc();        /* Install protected malloc and free function to ME */
IclPTInit(15);               /* Init the Icl PT server task with a priority of 15
*/

#ifdef HOST
/*
** Initialise the Cil
** Declare 4 buffers of 256 bytes each.
** These buffers are not used here - choose minimum sizes.
*/
CilInit(4,256);
#endif

TaskResume(TaskCreate(-1,task, NULL, 14, 4096)); /* Start task */

while(1==1); /* loop */
}

/*****
*
*   Function : SignalHandler
*   Args      : UINT32 Signals
*
*   Description:
*       Signals          Signals raised by host
*
*   SignalHandler will be called when host raises signal **
*   Return Values:
*       None
*
*****/

void
SignalHandler(UINT32 Signals)
{
    if ((Signals & HostRequestBitMask) !=0)
    {

```

```

    /*
    ** Host has requested a block of data
    */
    /* TaskSignalSema(Semaphore); */
    Semaphore = 1;
}
/*
** Ignore any other signals raised - they're not for us
*/
}

/* Initialize the adaptive bit allocation tables on the PPs */

void
init_alloc_table(UINT32 index)
{
    int t_alloc;
    int p_alloc;
    int r_alloc;
    int k,lp;

    t_alloc = YSIZE*XSIZE/comp_table[index] - headersize;
    p_alloc = t_alloc/((AY*AX)/6);
    r_alloc = t_alloc%((AY*AX)/6);

    for(k=0;k<((AY*AX)/6);k++)
        local_alloc[k] = p_alloc;

    lp = 0;
    while (r_alloc>0)
    {
        local_alloc[lp++] ++;
        r_alloc--;
    }
}

void
init_alloc_table2(UINT32 index)
{
    int t_alloc;
    int p_alloc;
    int r_alloc;
    int k,lp;
    int bytes_per_frame;

    bytes_per_frame = index;

    t_alloc = bytes_per_frame - headersize;
    p_alloc = t_alloc/((AY*AX)/6);
    r_alloc = t_alloc%((AY*AX)/6);

    for(k=0;k<((AY*AX)/6);k++)
        local_alloc[k] = p_alloc;

    lp = 0;
    while (r_alloc>0)
    {
        local_alloc[lp++] ++;
        r_alloc--;
    }
}

```

# NAWCWD TP 8442

```

/*****
*   Description:
*
*   PCI/C80 MP linker file
*
*****/

-c
-heap 0x100000
-stack 0x10000
-l mp_cio.lib
-l \pcic80\lib\mp_task.lib
-l mp_int.lib
-l mp_rts.lib
-l mp_ptreq.lib
-l mp_ppcmd.lib
-l ppcmd.lib
-l \pcic80\lib\icl.lib
-l \pcic80\lib\vol.lib
-l \pcic80\lib\bgl.lib
-l \pcic80\lib\vil24.lib
-l \pcic80\lib\cil.lib

MEMORY
{
    RAM0 : o=0x00000000    l = 0x00800
    RAM1 : o=0x00000800    l = 0x00800
    RAM2 : o=0x00008000    l = 0x00800
    RESERV : o=0x01000000    l = 0x00200
    MPPRAM1 : o=0x01010580    l = 0x00280
    MPPRAM : o=0x010007D8    l = 0x00028
    SDRAM : o=0x80000000    l = 0x300000
    DRAM : o=0x90000000    l = 0x80000
    DRAM2 : o=0x90080000    l = 0x100000
    UNINIT : o=0x90180000    l = 0x180000
    IMAGE : o=0x80300000    l = 0x80000
    SPOT : o=0x80380000    l = 0x10000
    VRAM_PAL : o=0xB0000000    l = 0x200000
    VRAM_VGA : o=0xB4000000    l = 0x400000
}

SECTIONS
{
    /*
    * The following section must be defined for all program that
    * use the CIL. The section must appear in the first 8Mb
    * of DRAM and must be long enough to include all buffers
    * plus 128 bytes. This example is big enough for 4*256byte
    * buffers.
    *
    * See the user guide for more information.
    */
    .lsidram : {
        _CilDRAMBase = .;
        . += 0x600;
    } > DRAM

    .text : > DRAM
    .cinit : > DRAM
    .const : > DRAM
    .switch : > DRAM
    .data : > DRAM
    .bss : > DRAM

```

## NAWCWD TP 8442

```
.cio:    > DRAM
.pcinit :    > DRAM

.ptext   :    load > DRAM2, run SDRAM

font:    load > DRAM2, run SDRAM

.systemem :    > UNINIT
.stack   :    > UNINIT
sh_vars  :          > MPPRAM
mp_vars  :          > MPPRAM1

rawimage:    > IMAGE
stream:      > SPOT
}
```

# NAWCWD TP 8442

```

/*****
 *
 *   Description:
 *
 *   PCI/C80 PP linker file
 *
 *****/
-pc
-l d:\mvp\src\newlib\pp_rts.lib

-pstack 192

MEMORY
{
    RAM0 : o=0x00000000   l = 0x00800
    RAM1 : o=0x00000800   l = 0x00800
    RAM2 : o=0x00008000   l = 0x00800
    RESERV : o=0x01000000   l = 0x00200
    PRAM : o=0x01000200   l = 0x00600
    SDRAM : o=0x80000000   l = 0x800000
    DRAM : o=0x90400000   l = 0x400000
    VRAM_PAL : o=0xB0000000   l = 0x200000
    VRAM_VGA : o=0xB4000000   l = 0x400000
}

SECTIONS
{
    .text      :   > DRAM
    .ptext     :   > DRAM
    .cinit     :   > DRAM
    .const     :   > DRAM
    .switch    :   > DRAM
    .data      :   > DRAM
    .bss       :   > DRAM
    .cio       :   > DRAM
    .system    :   > DRAM
    .stack     :   > DRAM

    .pcinit    :   > DRAM

    .pbss      :   (PASS) > PRAM
    .psystem   :   (PASS) > PRAM
    .pstack    :   (PASS) > PRAM
}

```

# NAWCWD TP 8442

```

/*****
**
** subpass.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutine:
**
** $subpass, $encode_symbol2, $update_model2, $bit_plus_follow2, I_DIV_JW2,
** $new_output_bit2
**
** The 2 at the end of the subroutine name indicates that these subroutines
** were copied here from another file and called locally to not have cache
** faults.
**
*****/

.global $subpass

.global $stats_flag      ;unsigned char  pointer *(xba + $stats_flag)
.global $char_to_index   ;unsigned char  pointer      &*(xba      +
$char_to_index)
.global $stats_val       ;signed   short  pointer *(xba + $stats_val)
.global $BITE            ;signed   int    sw *(xba + $BITE)
.global $STOP            ;signed   int    sw *(xba + $STOP)

.global $sym_index
.global $sym_array

.global $list
.global $list_index

.global $bit_index

.global $index_to_char
.global $getaway_address
.global $quick_getaway
.global $byte_stream
.global $No_of_symbols
.global $bits_to_follow
.global $T_BYTES
.global $high
.global $freq
.global $slow
.global $cum_freq

.global end_subpass3

tempd1      .setd4
tempd2      .setd2
tempd3      .setd5
tempd4      .setd1

m           .setd6
tt          .setd7

list        .seta12
stats_val   .seta4

i           .set20
j           .set24
t           .set28

```

```

.align 512

/*****
*
* Function : $subpass
* Args : t
* Passed in : d1
*
* Description:
*   t - the current THRESH >> BITE value.
*
*   $subpass will be called to perform a subordinate pass over
*   the coefficients.
*
* Return Values:
*   none
*****/

$subpass:

    d0 = &*(sp --= 32)
    *(sp + 12) =w iprs

    *(sp + 8) =w d6
||    *(sp + 16) =w a4

    *(sp + 4) =w a12
||    *(sp + 0) =w d7

    *(sp + t) = d1
    list =uw *(xba + $list)
    tempd1 =uh *(xba + $list_index) ;repeat loop list_index times

mloop:
    le0 = firstloop
    lrs0 = tempd1 - 1
    nop
    tempd2 =uh *(list++)

    stats_val =uw *(xba + $stats_val)

    tempd1 =uw *(xba + $BITE)
    tt = *(sp + t)
    || tempd2 = tempd2<<1
    le1 = endjloop
    lrs1 = tempd1 - 1

    stats_val = stats_val + tempd2
    tt = tt >>u 1

jloop:
;>>>>    sym = (stats_val[(s*768)+i]>0)&1;

;>>>>    stats_val[(s*768)+i] -= ((sym>0)?1:-1)*t/2;

    tempd2 =sh *(stats_val)
||    tempd1 = tt
    tempd2 = tempd2 - 0
    tempd4 =[.nvz] 1 || tempd4 =[le.nvz] zero

```

```

    tempd1 =[le] -tt

    tempd2 = tempd2 - tempd1
|| x1 = tempd4
    *(stats_val) =sh tempd2

    a1 = &*(xba + $char_to_index)

    call = $encode_symbol2
    d6 =ub *(a1 + x1)
    d1 = d6

    call = $update_model2
    nop
    d1 = d6

    d0 =sw *(xba + $STOP)
    d0 = d0 - 0
    br =[ne] done
    nop
    nop

endjloop:
    tt = tt >>u 1

firstloop:
    tempd2 =uh *(list++)

done:
    a12 =sw *(sp + 4)
    a4 =sw *(sp + 16)
    br = *(sp + 12)

    d6 =sw *(sp + 8)
    || d7 =sw *(sp + 0)
    d0 = &*(sp += 32)
;    branch occurs here

/*****
*
*   Function : encode_symbol2
*   Args      : none
*
*   Description:
*       encode_symbol will be called to perform the arithmetic
*       encoding of a symbol. This routine was lifted from a C
*       compiled program and included here, for cache coherency.
*
*   Return Values:
*       None
*
*****/

$encode_symbol2:
    x0 = d1
    a0 = &*(xba + $cum_freq)
    d3 =sw *(xba + $low)
    d2 =sw *(xba + $high)

```



```

    d1 = x0 << 1

    d4 = d2 - d3
||    d2 =g  a0

    a1 = d1 + d2
||    d0 = &*(sp --= 4)

    d4 = d4 + 1
||    *(sp) =w  iprs

    d1 =uh1  d4
||    d5 =sh  *(a1 - 2)

    d1 =u d5 * d1
||    d2 =uh1  d5
    d2 =u d2 * d4

    d1 =u d5 * d4
||    d2 = d2 + d1
    call = I_DIV_JW

    d1 = d1 + (d2 << 16)
||    x1 =sh  *a0
    d2 = x1

    d0 =uh1  d4
||    d1 =sh  *(a0 + [x0])

    d0 =u d0 * d1
||    d2 =uh1  d1
    d2 =u d4 * d2

    d1 =u d4 * d1
||    d2 = d0 + d2
    d1 = d1 + (d2 << 16)

    d4 = d5 + d3
||    d2 = x1
    call = I_DIV_JW
    d4 = d4 - 1
    *(xba + $high) =w  d4

    d2 = d5 + d3
    d1 = d4 - (1 \\ 8)
    br =[ge] L25
    *(xba + $low) =w  d2
    d1 =[ge] d2 - (1 \\ 8)

L19:
    call = $bit_plus_follow
    nop
    d1 = 0

    br = L23
    d1 =sw  *(xba + $high)
    d1 = d1 << 1

L20:
    d1 =sh  *(xba + $bits_to_follow)

    d2 = d1 + 1

```

```

||      d3 =sw  *(xba + $low)
        *(xba + $bits_to_follow) =h  d2

        d2 = d3 - (1 \\ 7)
||      d1 =sw  *(xba + $high)
        br = L22

        d1 = d1 - (1 \\ 7)
||      *(xba + $low) =w  d2
        *(xba + $high) =w  d1

L21:
        call = $bit_plus_follow
        nop
        d1 = 1

        d1 =sw  *(xba + $low)

        d1 = d1 - (1 \\ 8)
||      d2 =sw  *(xba + $high)

        d1 = d2 - (1 \\ 8)
||      *(xba + $low) =w  d1
        *(xba + $high) =w  d1
L22:
        d1 = d1 << 1
L23:

        d4 = d1 + 1
||      d1 =sw  *(xba + $low)
        d2 = d1 << 1
        d1 = d4 - (1 \\ 8)
        br =[lt] L19
        *(xba + $high) =w  d4
        *(xba + $low) =w  d2

L24:
        d1 = d2 - (1 \\ 8)
L25:
        br =[ge] L21
        nop
        d1 =[lt] d2 - (1 \\ 7)

        br =[lt] L30
        nop
        d1 =[ge] d4 - 384

        br =[lt] L20
        br =[ge] L30
        nop

        nop

L30:
        br = *(sp)
        nop
        d0 = &*(sp += 4)

```

# NAWCWD TP 8442

```

/*****
*
*   Function : I_DIV_JW2
*   Args      : none
*
*   Description:
*       I_DIV_JW will be called to perform an Integer Divide.
*
*   Return Values:
*       None
*
*****/

;*****
;* I_DIV.ASM   v1.10   - Integer Divide
;* Copyright (c) 1993-1995 Texas Instruments Incorporated
;*****

; +-----+
; |       i_div.asm = PP assembly program that is used to return a 32-bit |
; |       signed integer quotient from 32-bit signed integer           |
; |       division when called by a C program.                         |
; +-----+

        .global    I_DIV_JW2

; +-----+
; | 32-bit Signed Integer Word Divide Subroutine :                     |
; |  o Input 32-bit signed integer Operand 1 is in d1 (numerator).      |
; |  o Input 32-bit signed integer Operand 2 is in d2 (divisor).        |
; |  o Output 32-bit signed integer is in d5 (Answer = quotient).       |
; |  o Output 32-bit signed remainder is discarded.                    |
; |  o 0 input divisor produces 0x80000000 output with overflow set.    |
; |  o Quotient = 0x80000000 sets overflow.                             |
; |  o Number of Stack Words used = 3.                                  |
; |  o MF register is saved.                                           |
; |  o NOTE: Loop Counter 2 Registers are used but NOT restored !     |
; +-----+

; +-----+
; |       32 bit / 32 bit ==> 32 bit signed quotient                   |
; |       Signed PP Integer Division                                   |
; |       Numerator / Denominator = Quotient + Remainder (discarded)    |
; |       Divide by 0 produces 80000000 and sets sr(V)                 |
; |       Divide Overflow is not possible if Divisor is non-zero,      |
; |       except 80000000/ffffffff = 80000000 will set sr(V).          |
; |       MF register is preserved.                                     |
; +-----+

; .ptext      ; PP assembly code

arg1:  .setd1  ; input argument 1 = Numerator (32 low bits)
arg2:  .setd2  ; input argument 2 = Divisor (32 bits)
ans:   .setd5  ; answer = 32 bit signed quotient
Div:   .setd3  ; Input Divisor
Num:   .setd4  ; Input high Numerator = 0
Tmp:   .setd5  ; ALU output for each DIVI

; .align 8*16; start on a 16-instruction boundary

```

# NAWCWD TP 8442

I\_DIV\_JW2: ; Signed Word Integer Divide: Ans = Op1 / Op2

```

    Div = 0 - | arg2 |          ; negate | divisor |
    || *(sp--[3]) = Div        ; || push Div
    br = [z] Div_By_0          ; Divide By 0 ?
    Num = 0                    ; high numerator = 0
    || *(sp+[1]) = mf          ; || push mf
    || *(sp+[2]) = Num         ; || push Num
    mf = | arg1 |              ; input lo | numerator |

    lrse2 = 29                  ; loop count - 1
    Tmp = divi(Div, Num=Num)    ; 1-st divide iterate
    Tmp = divi(Div, Num=Tmp [n] Num) ; 2-nd divide iterate
LoopSW: Tmp = divi(Div, Num=Tmp [n] Num) ; divide iterate 3-32

    ans = mf                    ; | ans | = mf
    || Div = *sp++              ; || pop Div
    Num = arg1 ^ arg2           ; quotient sign
    || br = iprs                ; || return
    ans =[n] -ans               ; quotient is negative,
    || mf = *sp++              ; || pop mf
    Num = *sp++                 ; pop Num

Div_By_0: ; Divide By 0 \_____ Optional Error
Div_Ovfl: ; Divide Overflow /_____ Return Code

    br = iprs                  ; return
    || Div = *sp++             ; || pop Div
    mf = *sp++                 ; pop mf
    ans = 0 - 1<<31            ; returns 0x80000000, sets sr(V)
    || Num = *sp++             ; || pop Num ...[END]

```

.global \$update\_model

```

/*****
*
* Function : update_model2
* Args      : none
*
* Description:
*   update_model will be called to update the arithmetic
*   model's parameters. This routine was lifted from a C
*   compiled program and included here, for cache coherency.
*
* Return Values:
*   None
*
*****/

```

```

$update_model2:
    a0 = &(xba + $cum_freq)
    nop

    a1 = d1
    || d1 =sh *a0
    d1 = d1 - 75
    br =[ne] L6
    nop
    a2 =g [ne.ncvz] a1

```

# NAWCWD TP 8442

```

    d2 =sw *(xba + $No_of_symbols)
    d1 = d2 - 0
    br =[lt] L6
nop
    a2 =g [lt.ncvz] a1

    d1 =g a0
    le1 = L5 - 8

    d5 = d2 << 1
||    d3 = &*(xba + $freq)
    a0 = d5 + d1

    d4 = d2 << 1
||    lrs1 = d2
    a8 = d4 + d3
    d2 = 0

L4:
    d1 =sh *a8
    d1 = d1 + 1
    d3 = d1 >>u 31
    d1 = d1 + d3
    d1 = d1 >>s 1
    d1 =sh0 d1
    *a8 =h d1
    *a0-- =h d2
    d1 =sh *a8--
    d2 = d2 + d1

L5:
    a2 =g a1

L6:
    d3 = &*(xba + $freq)
    d1 = a2 << 1
    d5 = a2 << 1
    d4 = d3 + d1
    a0 = d5 + d3
    a8 = d4 - 2
nop

    d4 =sh *a8
||    d3 =sh *a0
    d3 = d3 - d4
    br =[ne] L9
    d2 =g [ne.ncvz] a2
    d3 =[ne] d2 - a1

L7:

    d1 = d1 - 2
||    d3 =sh *--a8

    a2 = a2 - 1
||    d4 =sh *--a0
    d3 = d4 - d3
    br =[eq] L7
nop
nop

```

```

L8:      d2 =g  a2
        d3 = d2 - a1
L9:      br =[ge] L11
        nop
        nop

        x0 = &*(xba + $index_to_char)
        nop
        a8 =ub  *(a2 + x0)
        x8 = &*(xba + $char_to_index)
        a9 =ub  *(a1 + x0)
        *(a2 + x0) =b  a9
        *(a1 + x0) =b  a8
        *(a8 + x8) =b  a1
        *(a9 + x8) =b  a2
L11:     d3 =sh  *a0
        d3 = d3 + 1

        d3 = a2 - 0
        ||      *a0 =h  d3
        br =[le] L15
        br =[le] L16
        nop

        le2 = L14 - 8
        d2 = a2 - 1
        d3 = &*(xba + $cum_freq)
        lrs2 = d2
        a0 = d1 + d3
        nop

L13:     d1 =sh  *--a0
        d1 = d1 + 1
        *a0 =h  d1

L14:     br = L16
        nop
L15:     nop
        nop
L16:     br = iprs
        nop
        nop

.global $bit_plus_follow

```

# NAWCWD TP 8442

```

/*****
*
*   Function : bit_plus_follow2
*   Args      : none
*
*   Description:
*       bit_plus_follow will be called to output several bits that
*       have been encoded by the arithmetic encoder.
*
*   Return Values:
*       None
*
*****/

$bit_plus_follow2:
    d0 = &(sp --= 8)
    *(sp + 4) =w iprs
    call = $new_output_bit
    nop

    d6 = d1
||    *(sp + 0) =w d6

    d1 =sh *(xba + $bits_to_follow)
    d1 = d1 - 0
    br =[le] L36
    nop
    d1 =[gt] d6 - 0

    d6 = 1 || d6 =[ne] a15
L34:    call = $new_output_bit
    nop
    d1 = d6

    d1 =sh *(xba + $bits_to_follow)
    d1 = d1 - 1
    d1 =sh0 d1

    d1 = d1 - 0
||    *(xba + $bits_to_follow) =h d1
    br =[gt] L34
    nop
    nop

L36:    br = *(sp + 4)
    nop

    d6 =sw *(sp + 0)
||    d0 = &(sp += 8)

.global $bit_index

```

```

/*****
*
*   Function : new_output_bit2
*   Args      : d1 - the bit to append
*
*   Description:
*       new_output_bit will be called to append a single bit to
*       the bitstream array.
*
*   Return Values:
*       None
*
*****/

$new_output_bit2:
    d2 =sw  *(xba + $STOP)
    d2 = d2 - 1
    br =[eq] iprs
    d2 =uh  *(xba + $bit_index)
    d4 = d2 >>u 3
||    d3 =uw  *(xba + $byte_stream)

    a0 = d4 + d3
    d0 =uh  *(xba + $T_BYTES)

    d4 = (d2&7)
||    d3 =ub  *a0
    d3 = d3 | (d1 << d4)
    *a0 =b  d3
||    d5 = d2 + 1

    *(xba + $bit_index) =h  d5

    d5 = d5 - (d0 << 3)

    br =g iprs
    d1 = 1 || d1 =[lt] a15      ;calculate new STOP value
    *(xba + $STOP) =w  d1

end_subpass3:
    nop

```



# NAWCWD TP 8442

```

/*****
**
** ztr.s (PP Program)
**
** Written by Jim Witham, Code 472300D, 939-3599
**
** File that contains the following assembly language subroutines:
**
** $comp_ztr
**
*****/

.global $comp_ztr

/*****
*
* Function : $comp_ztr
* Args : s
* Passed in : d1
*
* Description:
* s - the subblock to process
*
* $comp_ztr will be called to calculate the ztl array for a
* single subblock.
*
* Return Values:
* None
*
*****/

$comp_ztr:
    lctl = 0x0 ;reset looping capability

    d2 = d1 << 9
    x0 = d2 + 0x1fe
    nop
    a0 = &(dba + x0) ;stats_val[s][m=255] = stats_val + (s*256 + m) *(2
bytes/word)
; (0x1fe, 0x3fe, 0x5fe, 0x7fe, 0x9fe, 0xbfe)

    d2 = d1 << 7
    x0 = d2 + 0xc7e
    nop
    a2 = &(dba + x0) ;ztl[s][p=63] = ztl + (s*64 + p) *(2 bytes/word)
    nop ; (0xc7e, 0xcfe, 0xd7e, 0xdfe, 0xe7e, 0xefe)
    a8 = a2

;clear out ztl[s][0..63] array
    x0 = d2 + 0x0c00
    nop
    a1 = &(dba + x0)
    lrse0 = 31
    d1 = 0
    nop

    *(a1++=[1])= d1

    lr0 = 3 ;first innerloop
    lr1 = 3 ;second innerloop
    lr2 = 11 ;outerloop

```

# NAWCWD TP 8442

```

le0 = firstloopend2
ls0 = firstloopstart2
le1 = secondloopend2
ls1 = secondloopstart2
le2 = outerloopend2
ls2 = outerloopstart2

nop
lctl = 0xba9 ;associate le0 with lc0, le1 with lc1, and le2 with lc2

nop
nop

d4 =sh *(a0--[1])

outerloopstart2:
    nop

firstloopstart2:

    d4 = |d4|
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    || d3 =sh *(a0--[1])

    d3 = |d3|
    d3 = 31 - lmo(d3)
    d3 = 1<<d3

    d2 = d2 | d4 | d3
    || d4 =sh *(a0--[1])

firstloopend2:
    *(a2--[1]) =h d2

    a7 =uh *(a2++=[4])

secondloopstart2:

    d4 = |d4|
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    || d3 =sh *(a0--[1])

    d3 = |d3|
    d3 = 31 - lmo(d3)
    d3 = 1<<d3

    d2 = d2 | d4 | d3
    || d4 =sh *(a0--[1])

secondloopend2:
    *(a2--[1]) =h d2

outerloopend2:
    nop

lr0 = 1      ;first innerloop
lr1 = 1      ;second innerloop
lr2 = 5      ;outerloop

```

```

le0 = firstloopend3
ls0 = firstloopstart3
le1 = secondloopend3
ls1 = secondloopstart3
le2 = outerloopend3
ls2 = outerloopstart3

```

```

nop
nop

```

```

outerloopstart3:
    nop

```

```

firstloopstart3:

```

```

    d4 = |d4|
    || d3 =h *(a8)
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
    || d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
    || *(a8--[1]) =h d3

```

```

    d4 = |d4|
    || *(a2) =h d2
    d4 = 31 - lmo(d4)
    || d3 =h *(a8)
    d4 = 1<<d4
    || d2 =h *(a2)
    d3 = d3 | d4
    || d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
    || *(a8--[1]) =h d3

```

```

firstloopend3:
    *(a2--[1]) =h d2

```

```

    a7 =uh *(a2+=[2])

```

```

secondloopstart3:

```

```

    d4 = |d4|
    || d3 =h *(a8)
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
    || d4 =sh *(a0--[1])

```

```

    d2 = d2 | d3
    || *(a8--[1]) =h d3

```

```

    d4 = |d4|
    || *(a2) =h d2
    d4 = 31 - lmo(d4)
    || d3 =h *(a8)
    d4 = 1<<d4

```

```

    || d2 =h *(a2)
    d3 = d3 | d4
    || d4 =sh *(a0--=[1])

    d2 = d2 | d3
    || *(a8--=[1]) =h d3

secondloopend3:
    *(a2--=[1]) =h d2

outerloopend3:
    nop

    lr0 = 2      ;first loop

    le0 = firstloopend4
    ls0 = firstloopstart4

    nop
    nop

firstloopstart4:

    d4 = |d4|
    || d3 =h *(a8)
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
    || d4 =sh *(a0--=[1])

    d2 = d2 | d3
    || *(a8--=[1]) =h d3

    d4 = |d4|
    || *(a2) =h d2
    d4 = 31 - lmo(d4)
    || d3 =h *(a8)
    d4 = 1<<d4
    || d2 =h *(a2)
    d3 = d3 | d4
    || d4 =sh *(a0--=[1])

    d2 = d2 | d3
    || *(a8--=[1]) =h d3

    *(a2) =h d2

    d4 = |d4|
    || d3 =h *(a8)
    d4 = 31 - lmo(d4)
    || d2 =h *(a2)
    d4 = 1<<d4
    d3 = d3 | d4
    || d4 =sh *(a0--=[1])

    d2 = d2 | d3
    || *(a8--=[1]) =h d3

    d4 = |d4|

```

# NAWCWD TP 8442

```

    || *(a2) =h d2
d4 = 31 - lmo(d4)
    || d3 =h *(a8)
d4 = 1<<d4
    || d2 =h *(a2)
d3 = d3 | d4
    || d4 =sh *(a0--=[1])

d2 = d2 | d3
    || *(a8--=[1]) =h d3

*(a2--=[1]) =h d2

firstloopend4:
    nop

d4 = |d4|
    || d3 =h *(a8)
d4 = 31 - lmo(d4)
    || d2 =h *(a2)
d4 = 1<<d4
d3 = d3 | d4
    || d4 =sh *(a0--=[1])

d2 = d2 | d3
    || *(a8--=[1]) =h d3

d4 = |d4|
    || *(a2) =h d2
d4 = 31 - lmo(d4)
    || d3 =h *(a8)
d4 = 1<<d4
    || d2 =h *(a2)
d3 = d3 | d4
    || d4 =sh *(a0--=[1])

d2 = d2 | d3
    || *(a8--=[1]) =h d3

*(a2) =h d2

d4 = |d4|
    || d3 =h *(a8)
d4 = 31 - lmo(d4)
    || d2 =h *(a2)
d4 = 1<<d4
d3 = d3 | d4
    || d4 =sh *(a0--=[1])

d2 = d2 | d3
    || *(a8--=[1]) =h d3

d4 = |d4|
    || *(a2) =h d2
d4 = 31 - lmo(d4)
    || d3 =h *(a8)
d4 = 1<<d4
    || d2 =h *(a2)
d3 = d3 | d4

```

# NAWCWD TP 8442

```
d2 = d2 | d3  
|| *(a8) =h d3
```

```
*(a2) =h d2
```

```
br = iprs  
nop  
nop
```

## APPENDIX C

### LIST OF PUBLICATIONS AND PATENTS

#### PATENTS

1. C. D. Creusere, Ridgecrest, CA. Parallel Digital Image Compression System Which Exploits Zerotree Redundancies in Wavelet Coefficients. Submitted to U.S. Patent Office May 1998.
2. C. D. Creusere, Ridgecrest, CA. Efficient Embedded Image and Video Compression Using Lifted Wavelets. Submitted to U.S. Patent Office March 1999. (NC no. 79393.)

#### JOURNAL PAPERS

1. C. D. Creusere. "A New Method of Robust Image Compression Based on the Embedded Zerotree Wavelet Algorithm," *IEEE Trans. on Image Processing*, Vol. 6, No. 10 (October 1997), pp. 1436-1442.
2. C. D. Creusere. "Fast Embedded Compression for Video," accepted by *IEEE Trans. on Image Processing* for December 1999 publication.
3. C. D. Creusere. "Parallel Image Compression Which Exploits Inter-Scale Correlation in Wavelet Coefficients," submitted to *IEEE Trans. on Parallel & Distributed Systems*, November 1997; revised September 1998.
4. C. D. Creusere. "Motion Compensated Video Compression With Reduced Complexity Encoding for Remote Transmission," submitted to *Signal Processing: Image Communications*, May 1999.

#### CONFERENCE PAPERS

1. C. D. Creusere. "Image Coding Using Parallel Implementations of the Embedded Zerotree Wavelet Algorithm," *Proc. of the Digital Video Compression Conf. (Algorithms and Technologies 1996)*, pp. 82-93.
2. C. D. Creusere. "A Family of Image Compression Algorithms Which Are Robust to Transmission Errors," *Proc. SPIE*, Vol. 2825 (August 1996), pp. 890-900.
3. C. D. Creusere. "Out-of-Loop Motion Compensation for Reduced Complexity Video Encoding," *Proc. Data Compression Conf.*, pp. 428, and *Data Compression Industry Workshop*, pp.28-37 (March 1997).

4. C. D. Creusere. "Periodic Pan Compensation for Reduced Complexity Video Compression," *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing*, Vol. IV (April 1997), pp. 2889-2892.
5. C. D. Creusere. "A New Approach to Global Motion Compensation Which Reduces Video Encoding Complexity," *Proc. Int. Conf. on Image Processing*, Vol. III (October 1997), pp. 634-637.
6. C. D. Creusere. "Adaptive Embedding for Reduced Complexity Image and Video compression," *Proc. of the SPIE*, Vol. 3309, Visual Communications and Image Processing (January 1998), pp. 48-57.
7. C. D. Creusere. "Fast Embedded Video Compression Using Cache-Based Processing," *Proc. of the European Signal Processing Conf.* (September 1998).



## INITIAL DISTRIBUTION

- 1 Naval Air Systems Command, Patuxent River  
AIR-4.0T3, Shumway (1)
- 5 Chief of Naval Research, Arlington
  - ONR-311, Lau (1)
  - ONR-311, Masters (1)
  - ONR-311, Quinn (1)
  - ONR-351, Chew (1)
  - ONR-351, Siegal (1)
- 1 Naval Surface Warfare Center Dahlgren Division, Dahlgren (B05, Staton)
- 1 Naval War College, Newport (Technical Library)
- 1 Headquarters 497 IG/INT, Falls Church (OUWG Chairman)
- 2 Defense Technical Information Center, Fort Belvoir
- 1 Cognitech, Inc., Pasadena, CA (Rudin)
- 2 Mathsoft Data Analysis Product Division, Seattle, WA
  - Bruce (1)
  - Goldschneider (1)
- 2 Raytheon Systems Company, Tucson, AZ
  - Mahalanobis (1)
  - Miller (1)
- 2 University of California Los Angeles, Los Angeles, CA
  - Osher (1)
  - Chan (1)
- 2 University of South Carolina, Columbia, SC
  - De Vore (1)
  - Sharpely (1)

---

## ON-SITE DISTRIBUTION

- 1 Code 4BT000D, Markarian
- 1 Code 4TB000D, Fischer
- 4 Code 4TL000D (3 plus Archives copy)
- 1 Code 4T1000D, Loftus
- 11 Code 4T4400D
  - Schwartz (1)
  - Creusere (10)
- 1 Code 45T000D, Tanaka
- 1 Code 452330D, Armogida
- 1 Code 452350D, Stone
- 1 Code 47BE00D, Burdick
- 1 Code 471000D, Mello
- 12 Code 471600D
  - Hanson (1)
  - Hewer (10)
  - Van Nevel (1)
- 5 Code 472300D, Witham
- 1 Code 473200D, Ghaleb